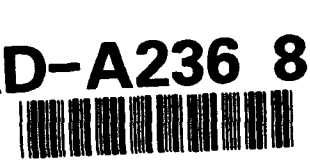

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A Specification and Analysis of the
IEEE Token Bus Protocol

Lauren J. Charbonneau
June 1990

Thesis Advisor: G. M. Lundy

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: Computer Science Dept. Naval Postgraduate School; 6b. OFFICE SYMBOL (if applicable): 52	7a. NAME OF MONITORING ORGANIZATION: Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943	7b. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: Naval Postgraduate School; 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code): Monterey CA 93943	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
A Specification and Analysis of the IEEE Token Bus Protocol

12. PERSONAL AUTHOR(S)
Charbonneau, Lauren J.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis	FROM 09/89 TO 06/90	June 1990	87

16. SUPPLEMENTARY NOTATION
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
token bus protocol, *systems of communicating machines*, systems state analysis

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

In this thesis a formal description technique, *systems of communicating machines*, is used to specify and analyze a token bus protocol. A simplified description of the protocol is given, and proofs of certain correctness properties presented. The analysis proves that the protocol is free from deadlocks and nonexecutable transitions, and also that successful message transfer is guaranteed for a network with an arbitrary number of machines. A program written in an object oriented language, C++, demonstrates that the description technique, the specification, and the analysis of the protocol is complete and accurate for a network of three stations. The specification is then extended to allow the transmission of different types of messages, errors in the communication channel, acknowledgments from the receiver, and timeouts.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: [X] UNCLASSIFIED/UNLIMITED [ ] SAME AS RPT. [ ] DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: G. M. Lundy	22b. TELEPHONE (Include Area Code): (408) 646-2094; 22c. OFFICE SYMBOL: 52LN

Approved for public release; distribution is unlimited

A SPECIFICATION AND ANALYSIS of the IEEE TOKEN BUS PROTOCOL

by
Lauren J. Charbonneau
Lieutenant , United States Navy
B.B.A., University of Texas, 1985

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1990

Author: Lauren J. Charbonneau

Approved By: G.M. Lundy, Thesis Advisor

Man-Tak Shing, Second Reader

Robert B. McGhee, Chairman,
Department of Computer Science

Abstract

In this thesis a formal description technique, *systems of communicating machines*, is used to specify and analyze a token bus protocol. A simplified description of the protocol is given, and proofs of certain correctness properties presented. The analysis proves that the protocol is free from deadlocks and nonexecutable transitions, and also that successful message transfer is guaranteed for a network with an arbitrary number of machines. A program written in an object oriented language, C++, demonstrates that the description technique, the specification, and the analysis of the protocol is complete and accurate for a network of three stations. The specification is then extended to allow the transmission of different types of messages, errors in the communication channel, acknowledgments from the receiver, and timeouts.

TABLE OF CONTENTS

I. INTRODUCTION

A. FORMAL MODELING OF PROTOCOLS

Communication protocols are the parallel algorithms which provide the means for communication between computers connected in networks. These protocols exist at every level in computer networks, from the physical level, where strict procedures must be followed for the correct transfer of signals from one machine to the next, to the application level, which involves the passing of text messages from a software program in one machine to another, or the transfer of an electronic mail message. Most of these protocols are rather complex, involving the synchronization in one form or another of programs in various autonomous machines. As the nucleus of teleprocessing networks, protocols are responsible for ensuring that these autonomous machines operate as a cohesive system. Therefore, the need exists for documentation to be written in such a way that the details are easily interpretable and unambiguous to all concerned parties. It is essential that protocols are clearly described, without ambiguity, and that they function correctly, accomplishing their intended function, without errors.

The importance of a clear description, and of analysis, to show that protocols function correctly, has led to much research in the past decade. Several methods have been suggested for formal modeling.

One of the early popular methods modeled protocols with event driven processes that communicated with each other through message passing. These processes were represented by finite state machines; thus the term *communicating finite state machines*, [4,20]. In fact, each individual machine (station) in the network was described as a finite state machine, with the communication channels treated as FIFO queues. The

dynamics of the system were described in terms of global states and transitions between those global states. Petri Nets have also been used as a formal method for describing protocols.

Both of these methods used an analysis technique called *reachability analysis*, in which the set of all global states (a composite state, consisting of the state of all machines and queues in the network) was generated. For large protocols this method proved to be very cumbersome. Because of the combinatorial explosion in the number of states, some protocols could not be formally specified and analyzed as a whole entity. Analysis had to be performed on subsets, if at all.

Another model [3] treated each machine as a process that was described with program language statements. Specialized programming languages have also been developed. In recent years much work has been done on two of these languages, LOTOS and Estelle [2,11]. In [1], Bochmann combined finite state machines with variables to describe protocols, uniting some elements of both models. The model utilized here, *systems of communicating machines*, carries that work further. It has been formally defined [12], making a formal analysis possible.

In this thesis, the TOKEN BUS protocol[8] is formally specified and an analysis is given. The analysis shows that the protocol is free from deadlocks and nonexecutable transitions, and that the protocol is *live*, which means that successful data transfer is guaranteed under the assumptions of the model. The initial specification has some simplifying assumptions which serve to make the main ideas of both the protocol and its analysis easily understood. The formal description technique used, called *systems of communicating machines*, is powerful enough to give a concise and simple description of the protocol. It also has an analysis method which is easily understood, called *system state analysis*. Thesis workups included a separate *system state analysis* on the protocol specification for networks that had as few as two and as many as ten stations. These lead eventually to a more general proof for an arbitrary number of machines.

The token bus protocol is one of three local area network protocols chosen by the IEEE for standardization; the others are the *Carrier Sense Multiple Access with Collision Detection* (CSMA/CD) [7] and the *Token Ring* [9] protocols. Both have been specified and analysed in previous work [15,18] using methods similar to those employed here.

Brief descriptions of two well known pre-standardization protocols are now presented. These are followed by more elaborate descriptions of three standardized local area network protocols. Chapter 2 then defines the formal model, *systems of communicating machines.* The specification of the protocol is then given in Chapter 3, and its analysis is in Chapter 4. Chapter 5 presents a working program written directly from the Chapter 3 specification. It is followed in Chapter 6 with an extension that eliminates some of the previous simplifications. Finally, the thesis is summarized in Chapter 7.

B. ALOHA

Originating at the University of Hawaii, pure ALOHA was one of the first protocols to be used to link computers in a network. The basic idea of a pure ALOHA system was simple: users transmit on a broadcast medium whenever they have traffic to send. Collisions and lost messages will occur when two or more messages from two or more stations try to occupy the medium at the same time. When this occurs the transmitting stations simply wait a random period of time and transmit again. It is important to realize that if the first bit of a new message overlaps with the last bit of a message that is almost finished, both will become unintelligible and have to be retransmitted later. Both transmitting station's messages have now been delayed. Through previous research that assumed a Poisson arrival rate, it has been determined that in a network where each station is equally likely to begin transmitting at any time, the best possible channel utilization is approximately 18 percent [22].

Even if time is divided into discrete intervals and stations are only permitted to begin transmitting at the beginning of a time slot, the best utilization possible is 36

percent [22]. This slotted ALOHA scheme doubles the utilization of pure ALOHA but still leaves much to be desired. Another disadvantage is that there is no provision for different priority messages.

These types of access methods are best suited to LANs with very low levels of traffic or to network situations where a single (or very few) transmitting stations need to communicate with many other stations that predominantly listen. This is not the case in most LANs of today.

C. CARRIER SENSE MULTIPLE ACCESS (CSMA)

CSMA protocols require stations with traffic to listen to the communication medium to determine if any other station is transmitting before it begins transmitting its messages. If the medium is busy, the station waits (with varying degrees of persistence) until it becomes idle before transmitting. If a collision occurs, all transmitting stations wait a random period of time and start all over again.

Not considering propogation delay, collisions will always occur when two or more stations become ready in the middle of a third station's transmission. Both will wait politely until the transmission ends and then begin transmitting simultaneously, resulting in a collision. This 1-persistent CSMA scheme continually senses the medium and transmits with 100 percent probability when the medium becomes quiet. Variations on this include:

non-persistent CSMA – where upon sensing a busy medium, the protocol waits a random period of time before even sensing the medium again.

p-persistent CSMA – where upon sensing an idle medium, the protocol transmits with probability p or defers transmitting with probability q, where $(q = 1 - p)$. This method applies a sophisticated probability scheme that 'acts', with probability q, as if collisions have occurred. The effect is a reduction in the number of real collisions and subsequent delays throughout the network.

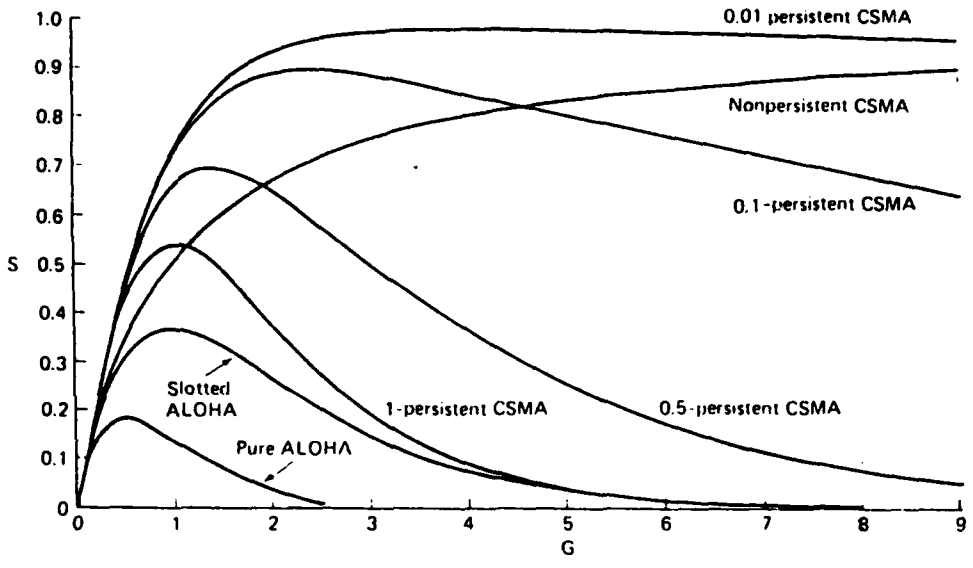

Figure 1: Comparison of Channel Utilization vs Load [22]

Figure 1 describes the throughput of all of the protocols mentioned so far. It is easy to see that the CSMA protocols are far better in terms of throughput than pure ALOHA because ready-to-transmit stations do not interfere with the station transmitting at the time they become ready. Intuitively, CSMA protocols provide higher performance than either pure or slotted ALOHA. However, delays can be considerable because transmitting stations always finish transmitting their messages even after a collision has been heard on the network. Useful bandwidth is lost finishing message transmissions that are known to be unintelligible. As with Aloha protocols, there is no provision for messages of different priority.

D. ANSI/IEEE STD 802.3 CARRIER SENSE MULTIPLE ACCESS WITH COLLISION DETECTION (CSMA/CD)

The IEEE CSMA/CD standard access method is an extension of the above mentioned CSMA. Like CSMA, it is a means by which two or more, (usually many more), stations share a passive broadcast transmission medium. It is commonly referred to as Ethernet.

This name has as its origin the 19th century hypothesis that luminiferous ether was the medium through which electromagnetic radiation propogated. This notion was long ago dispelled; however, the term Ethernet remains.

There is no central control in an Ethernet and access to the medium by stations needing to transmit is done in a distributed fashion, by the stations themselves, using the 1-persistent probability arbitration scheme. To transmit, a station waits until the medium is quiet (i.e., no other stations are transmitting), and then sends its message. If two or more stations begin transmitting at the same time, a collision will occur and all messages become unintelligible. If this occurs, all transmitting stations detect the collision, but unlike CSMA, transmit only a few additional jamming bytes to ensure propogation of the collision throughout the network. The stations then stop transmitting (without finishing their messages), wait a random period of time, listen for the medium to become quiet, and attempt to retransmit the same message again. The scheduling of the retransmissions is determined by a controlled randomization process called 'truncated binary exponential backoff'. The algorithms used to generate the random wait time for autonomous stations are designed to maximize the dispersion between wait times generated by any two stations at any given time [7].

Ethernets are far and away the most widely used LAN at present, with a huge installed base and considerable operational experience. The algorithm is simple and stations can be installed and removed without taking the network down. A passive cable is used and modems are not required. Furthermore, the delay at low load is practically zero, because stations do not have to wait for a time slot or token; they just transmit immediately. Another factor that enhanced the widespread acceptance of Ethernets was the fact that they were, and still are, inexpensive to implement. Economical network connections may very well be a more decisive factor than maximum bandwidth utilization for the propogation of useful data.

These types of LANs are best utilized in situations where most messages are over 1000 bytes in length and traffic is bursty and infrequent [22]. They are explicitly designed to have excess bandwidth, not all of which must be used, and systems operate most efficiently when engineered to run with a sustained load of less than 50 percent [21]. As a consequence of this, Ethernets will generally provide adequate throughput with low delay on lightly loaded networks. However, as the load increases, collisions increase, delays increase, and performance deteriorates rapidly [6,22]. In fact, there is no definitive upper bound on wait times. It is possible for two or more stations to repeatedly collide for an extended period of time. Again, there is no provision for different priority messages. These factors make CSMA/CD inappropriate for real-time systems, or for any network that has or anticipates high loads of traffic.

A formal specification and analysis of CSMA/CD using *systems of communicating machines* [15], has been completed. In that paper, Lundy proved the protocol to be free of deadlocks, but he also showed that the protocol was not *live*. That is, there was no guarantee that data transmission would be successful in a finite period of time.

E. TOKEN RING

The token ring standard access method is one of the first protocols to *NOT* utilize broadcast as a means to relay messages from transmitting to receiving stations. Rather, it uses a collection of individual point-to-point links that happen to form a closed unidirectional loop. A station must acquire the token, removing it from the ring, before it can transmit its messages. Once this occurs, the transmitting station will transmit its own outgoing message onto the ring. It is important to note that as each bit of this transmitted message arrives at each downstream station, it is read into a buffer and then, one bit time later, copied out onto the ring again. This one-bit time delay occurs at every station (or point) in the ring. The entire message will be relayed in this fashion from point-to-point and eventually return to the sender. The transmitting station

now has the responsibility of draining its message from the ring and can transmit more messages if time constraints permit. Once the token-holding-time window has passed, the token-holder must regenerate the token message and copy it onto the medium. The immediate downstream neighbor can now seize it and begin transmitting in exactly the same manner.

Only the token-holding transmitting station can send traffic. All other stations are forced to listen for traffic addressed to themselves, repeat bits, and wait for their turn to have the token and begin transmitting their own messages.

Not considering priority messages, the token ring is fair in the sense that all stations will, in a round-robin fashion, get their turn to access the medium and transmit. Unlike any of the previous protocols, it has a deterministic upper bound on channel access time. This is an attractive feature when compared to CSMA/CD. Because of the way that each station gets its turn, network throughput and efficiency can approach 100 percent under conditions of heavy load. It is also inexpensive and easy to install.

A disadvantage of the token ring is that when traffic is light, a station will still have to wait until it sees the token from its physical upstream neighbor before it can transmit. This delay will be at least as long as the time it takes for a station to complete the get token, check buffers, pass token sequence multiplied by $n-1$ stations in the ring. Added to this delay is the n bit times that are induced at each station when the transmitted frames are repeated throughout the network. Another criticism of token ring LANs is the fact that a down station anywhere in the loop will bring the whole network down. Along this same line, the use of a centralized network monitoring station induces network maintenance problems if that station becomes degraded or inoperative.

The token ring protocol does provide for priority messages but the fairness property mentioned earlier no longer holds when this feature is implemented. The priority scheme allows stations with higher priority messages to acquire the token more quickly. Stations

with only low priority messages can be prevented from receiving a low priority token and transmitting their traffic.

A formal specification and analysis on a simplified version of this protocol was accomplished [17]. This paper showed that the passing of the token, the transmittal of a data frame, and its receipt and acknowledgement are accomplished by the specification. However, because of the number of system states involved, (632 for an IEEE standard two station network[18]), modeling a network of three or more stations was not attempted. As with CSMA/CD, the specification was formalized using *systems of communicating machines.*

F. FIBER DISTRIBUTED DATA INTERFACE (FDDI)

FDDI [19] is an ANSI draft proposed standard for a 100 Mbit fiber-optic token ring local area network. Because of the recent improvements in light-wave technology, FDDI can offer much higher data rates than the capacity of the older technology networks.

Like the standard token ring, FDDI uses a collection of individual point-to-point links that form a closed loop. Stations cooperatively use timers to maintain a specified target token rotation time (TTRT) by using the observed network load to regulate the amount of time that a station may transmit. This TTRT is adjustable to facilitate the various requirements associated with different applications.

Every station is guaranteed a minimum token holding time (THT), all of which need not be used, each time the token is acquired. The traffic transmitted during these token holding periods is termed *synchronous.* From a particular station, if the revolving token returns before its TTRT timer has expired, that station will increase its allowable THT by the difference of the specified TTRT minus the actual time it took for the token to complete a loop through the network. The extra, less critical, traffic that can be transmitted during this interval is called *asynchronous.* By allowing both *synchronous* and *asynchronous* traffic, stations with heavy loads can transmit longer if there are sta-

tions that are not utilizing their full allotment. More efficient utilization of the physical bandwidth is realized.

It is important to note that there will be *asynchronous* traffic only if the *synchronous* traffic does not use the full target token rotation time in one cycle of the ring. In this case the TTRT timer will expire at each station before the token is reacquired.

The guarantee that the token will return within a specified time period enables FDDI to support applications that require guaranteed bandwidth, such as real time control and voice. In [10] it is formally proven that the timing requirements inherent in FDDI are satisfied when all components are functioning properly.

FDDI is expected to be the follow-on network to the current 802 LANs [19]. Other factors besides the aforementioned speed advantage are security, immunity to electromagnetic interference, and reduced weight and size. Optical fiber does not adapt well to bus configurations, hence the similarity to the token ring topology. Another advantage of FDDI is that its speed can allow it to be used as a backbone for bridges to a variety of other lower-speed LANs or as gateways to public data networks.

A formal specification and analysis of the protocol using *systems of communicating machines* was recently accomplished [14]. FDDI was analyzed for correctness properties and proof was given that it is free of deadlocks.

G. TOKEN BUS

The token bus protocol is a combination of both the CSMA/CD and token ring protocols. All stations on the network are connected by a single *bus* which, like CSMA/CD, functions as a broadcast medium. Physically the bus is a cable which propagates the signals transmitted by the stations throughout its entire length (see Figure 2). Any station may transmit bits onto the bus, and these bits will propagate to every other station on the network. However, only one station may transmit at a time; otherwise the signals

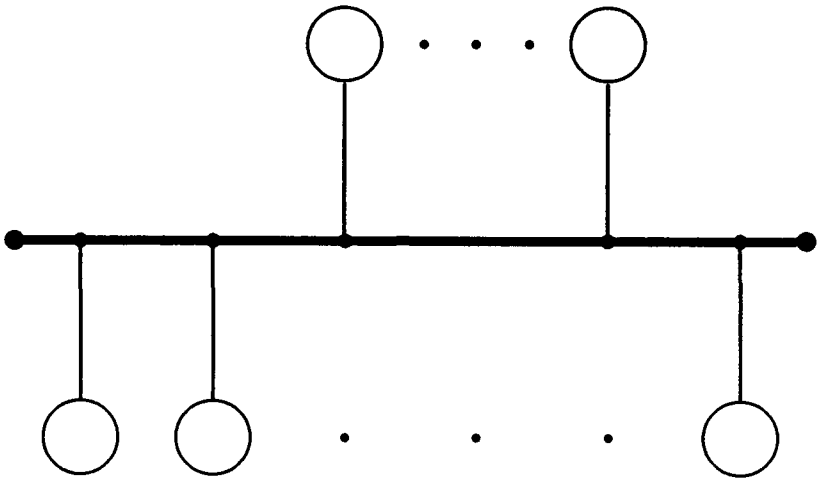

Figure 2: Topology of a Token Bus Network

would interfere, causing a *collision*. To ensure that a network has only one transmitting station, the protocol functions like the token ring and relays a token.

The *token*, which is carried as a unique message, is passed logically from station to station; only one token exists on the network, and only the station possessing it may transmit. Thus, access to the bus is limited to one station, so collisions are prevented. Like the token ring, a small percentage of the useful bandwidth is utilized for token passing and management, but none is required for collision detection and resolution.

If a station wishes to transmit a message to another station, it must wait until receiving the token. Upon receiving the token, it transmits the message, or *frame*, and then passes the token on to the next station. The stations on the network are ordered, so that each has an "upstream neighbor," from which the token is received, and

a "downstream neighbor," to which the token is passed. This ordering forms a cycle, so that the token periodically returns to each station.

In order to keep one station from taking control of the network (i.e., holding onto the token indefinitely), a limit must be placed on the token holding time. This can be implemented with a timer, or by limiting the number of frames a station may transmit before giving up the token.

The major advantage the token bus protocol has over the CSMA/CD protocol is that no time is lost due to collisions. This results in much better throughput under heavy loads. There is a deterministic upper bound on the time any station may have to wait to acquire the token and transmit a frame. This upper bound is essentially the product of the number of stations and the maximum token holding time. As previously noted, with CSMA/CD there is no such upper bound. In [5,6], comparisons of the two were shown based on simulation studies; similar results were reported.

Another advantage is the availability of four priority classes for outgoing message queues. When a station obtains the token, the highest priority queue immediately transmits its frames. If this queue empties and the station has not reached its maximum token holding time, the next highest priority queue will transmit its frames. It is easy to see that control cascades down the priority list until either the token holding station has transmitted all of its messages from all of its queues or the token holding time constraint has been met and the station passes the token to its neighbor. Some of the lower priority messages may remain where they are. The next pass of the token into this station starts with the highest priority messages again. Low priority messages will remain untransmitted until all higher priority messages from all higher priority queues have gone.

The highest priority queues at each station in a token bus network could be used to implement voice or other real time traffic [22]. This priority scheme can be implemented to *guarantee* a known fraction of the network bandwidth to the messages in the highest

priority queues. Because of its similarity in topology to Ethernets, any CSMA/CD Ethernet could quickly be converted to a much more efficient (at times of heavy load), real time capable token bus network.

The disadvantage of the token bus protocol, in comparison with CSMA/CD, is its increased overhead and complexity. Logical ring maintenance and token management are not trivial matters. Issues to be dealt with include loss of the token, duplicate tokens, and reestablishment of the logical ring after a station comes on-line or goes down. These issues are topics for future specification and analysis.

Advantages of the token bus over the token ring network are the simplicity of the topology and the above mentioned real time capable priority scheme. The token ring topology, however, seems more easily adapted to networks, such as FDDI, that utilize the increased performance of optical fiber technology.

II. SYSTEMS OF COMMUNICATING MACHINES

The formal description technique *Systems of Communicating Machines* [12], designed for the specification and analysis of communication protocols, uses a combination of finite state machines and variables to model a communication network. The communication between machines is accomplished through *shared variables*. Each machine also has *local variables* which may be used for various purposes, such as storing data blocks or for counters. *Enabling predicates* and *actions* are associated with each state transition; the enabling predicates determine when a transition may be taken, and the actions alter the variable values as the communication in the network progresses. The major method of analysis used with this model is called *system state analysis*. This is similar to the *reachability analysis* which has been used with other protocol models – especially the *communicating finite state machine* (CFSM) model – but provides an often significant reduction in the analysis. (See, for example, [16]). Other methods of analysis have also been used with this model; in [15], a proof method was used, which grouped sets of states together into classes.

Formally, *a system of communicating machines* is an ordered pair $C = (M, V)$, where

$$M = \{m_1, m_2, \ldots, m_n\}$$

is a finite set of *machines*, and

$$V = \{v_1, v_2, \ldots, v_k\}$$

is a finite set of *shared variables*, with two designated subsets R_i and W_i specified for each machine m_i. The subset R_i of V is called the set of *read access variables* for machine m_i, and the subset W_i the set of *write access variables* for m_i.

Each machine $m_i \in M$ is defined by a tuple $(S_i, s, L_i, N_i, \tau_i)$, where

(1) S_i is a finite set of states;

(2) $s \in S_i$ is a designated state called the *initial state* of m_i;

(3) L_i is a finite set of *local variables*;

(4) N_i is a finite set of names, each of which is associated with a unique pair (p, a), where p is a predicate on the variables of $L_i \cup R_i$, and a is an *action* on the variables of $L_i \cup R_i \cup W_i$. Specifically, an action is a partial function

$$a : L_i \times R_i \rightarrow L_i \times W_i$$

from the values contained in the local variables and read access variables to the values of the local variables and write access variables.

(5) $\tau_i : S_i \times N_i \rightarrow S_i$ is a transition function, which is a partial function from the states and names of m_i to the states of m_i.

Machines model the entities, which in a protocol system are processes and channels. The shared variables are the means of communication between the machines. Intuitively, R_i and W_i are the subsets of V to which m_i has read and write access, respectively. A machine is allowed to make a transition from one state to another when the predicate associated with the name for that transition is true. Upon taking the transition, the action associated with that name is executed. The action changes the values of local and/or shared variables, thus allowing other predicates to become true.

The set L_i of local variables specifies a name and a range for each. The range must be a finite or countable set of values.

A *system state tuple* is a tuple of all machine states. That is, if (M,V) is a system of n communicating machines, and s_i, for $1 \leq i \leq n$, is the state of machine m_i, then the n-tuple $(s_1, s_2, \ldots, s_n)$ is the system state tuple of (M,V). A *system state* is a system state tuple, plus the outgoing transitions which are enabled. That is, two system states are *equivalent* if every machine is in the same state, *and* the same outgoing transitions are enabled. The *initial system state* is the system state such that every machine is in its initial state, and the outgoing transitions are the same as in the initial global state.

The *global state* of a system consists of the system state, plus the values of all variables, both local and shared. It may be written as a larger tuple, combining the system

state with the values of the variables. The *initial global state* is the initial system state, with the additional requirement that all variables have their initial values. A global state *corresponds* to a system state if every machine is in the same state, and the same outgoing transitions are enabled. That is, a global state consists of a tuple of machine states, plus the values of all variables. A system state with the same tuple of machine states and the same enabled outgoing transitions is the corresponding system state.

Let $\tau(s_1, n) = s_2$ be a transition which is defined on machine m_i. Transition τ is *enabled* if the enabling predicate p, associated with name n, is true. Transition τ may be executed whenever m_i is in state s_1 and the predicate p is true (enabled). The *execution* of τ is an atomic action, in which both the state change and the action a associated with n occur simultaneously.

Note that if the values of all variables are restricted to some finite range, then the model can theoretically be reduced to a simple finite state machine. Otherwise, an infinite number of global states are possible. However, even if the number of global states is infinite, the number of system states is finite, because of the finiteness of each machine. This may allow a reachability analysis on the system states, when a reachability analysis on the global states is infinite. Even when the values of all variables are of a finite range, the number of global states in the equivalent FSM system may be so large as to be intractable.

Another advantage this model has over most other description techniques is in the modeling of communication channels. First, these are modeled through shared variables, which gives more flexibility than pure FIFO queues. Secondly, simultaneous transitions are allowed by the definition (unlike, for example, the CFSM model). These two advantages allow us to reasonably model local area networks using a bus as a communication medium. In [15], the CSMA/CD network was modeled, to include collisions. In [17] this model was used to specify and partially analyze the token ring protocol, and a complete system state reachability analysis for two machines was given in [18]. Work in conjunc-

tion with this thesis provided a complete system state analysis for a two machine token bus network [13]. Again, the *systems of communicating machines* model was utilized. Recent work is concentrating on the modeling of fiber optic networks, such as FDDI [14].

In this thesis, a shared variable, called *MEDIUM*, is used to model the common bus, which is the transmission medium. While this is an abstraction from the physical cable, it is felt that the use of a shared variable is a reasonable one, which allows for a realistic specification of the protocol. Obviously some abstraction is necessary and desirable.

The IEEE Standard 802.4 describes the token bus protocol. That standard uses a combination of finite state machines and the programming language Ada; however the model in this paper is precisely defined, and the communication channels are modeled by shared variables.

III. SPECIFICATION OF A TOKEN BUS PROTOCOL

This specification is a simplified one. It assumes that the transmission medium is error free and that all transmitted messages are received intact. The intent here is to portray the ideas of the protocol and to introduce the methodology of specification using *systems of communicating machines.* An extension that includes transmission of different types of data messages, acknowledgments, timeouts, and errors on the medium appears later in Chapter 6.

The specification of this simplified network consists of a specification for each machine, given in Figure 3 and Table 1, and the shared variable *MEDIUM*, also shown in Figure 3. This single shared variable, *MEDIUM*, is used to model the bus, which is "shared" by each machine. A transmission onto the bus is modeled by a write into the shared variable. The fields of this variable correspond to the parts of the transmitted message: the first field, *MEDIUM.t*, takes the values T or D, which indicate whether the frame is a token or a data frame. The second field contains the address of the station to which the message is transmitted (DA for "destination address"); the next field, the originator (SA for "source address"); and finally the data block itself.

The network stations, or machines, are defined by a finite state machine, a set of local variables, and a predicate-action table. The *initial state* of each machine is state 0, and the shared variable is initially set to contain the token with the address of one of the stations in the "DA" field.

The value of local variable *next* is the address of the next or downstream neighbor, and these are initialized so that the entire network forms a cycle, or logical ring.

The local variable *i* is used to store the station's own address. As implied by their names, the local variables *inbuf* and *outbuf* are used for storing data blocks to be

transmitted to or received from other machines on the network. The latter of these, *outbuf*, is an array and thus can store a potentially large number of data blocks. The variable *ctr* serves to count the number of blocks sent; it is an upper bound on the number of blocks which can be sent during a single token holding period. The local variable j is a pointer into the array *outbuf*.

The initial state of each machine is state 0, and local variables j and *ctr* are initially set to 1, and *inbuf* and *outbuf* are initially set to empty. The shared variable *MEDIUM* initially contains the token, with the address of one station in the DA field. Thus the initial system state tuple is (0,0,..,0) and the first transition taken will be *get-tk* by the station which has its local variable i equal to *MEDIUM.DA*.

Each machine has four states. In the initial state, 0, the station is quiescent, merely waiting to either receive a message from another station, or the token. If the token appears in the variable *MEDIUM* with the station's own address, the transition to state 2 is taken. When taking the *get-tk* transition, the machine clears the communication medium and sets the message counter *ctr* to 1. In state 2, the station transmits any data blocks it has, moving to state 3, or passes the token, returning to state 0. In state 3, the station will return to state 2 if any additional blocks are to be sent, until the maximum count k is reached. When the count is reached, or when all the station's messages have been sent, the station returns to state 0.

The receiving station, as with all stations not in possession of the token, will be in state 0. The message will appear in $MEDIUM$, with the receiving station's address in the DA field. The receiving transition to state 1 will then be taken, the data block copied, and the $MEDIUM$ cleared. By clearing the medium, the receiving station enables the sending station to return to its initial state (0) or to its sending state (2).

The reader may verify these transitions by examining the state diagram and the predicate-action table. The symbol "$\oplus$" indicates that the variable should be incremented unless its maximum value has been reached, in which case it should be reset to

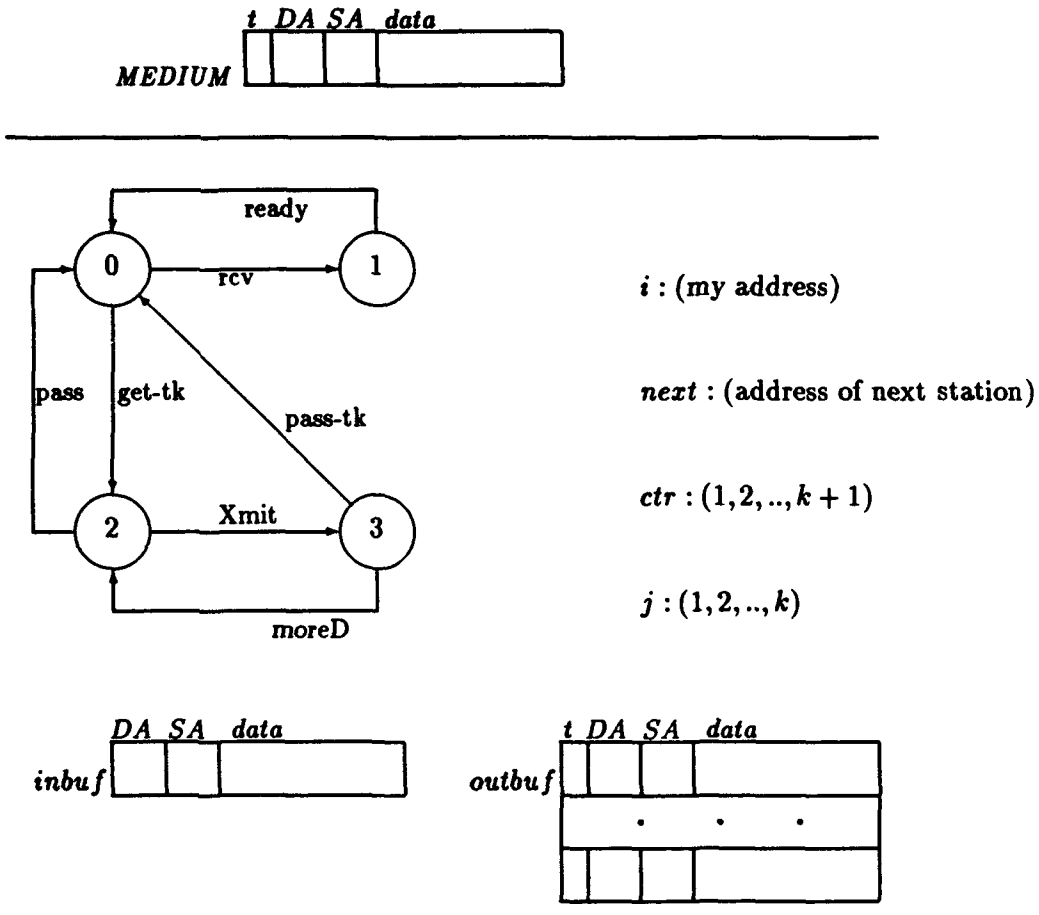

Figure 3: Specification of the Network Nodes

transition	*predicate*	*action*
rcv	$MEDIUM.(t, DA) = (D, i)$	$inbuf \leftarrow MEDIUM.(SA, data)$
ready	true	$MEDIUM \leftarrow \emptyset$
get-tk	$MEDIUM.(t, DA) = (T, i)$	$MEDIUM \leftarrow \emptyset$; $ctr \leftarrow 1$
pass	$outbuf[j] = \emptyset$	$MEDIUM \leftarrow (T, next, i, \emptyset)$
Xmit	$outbuf[j] \neq \emptyset$	$MEDIUM \leftarrow outbuf[j]$; $ctr \leftarrow ctr \oplus 1$; $j \leftarrow j \oplus 1$
moreD	$MEDIUM = \emptyset \wedge outbuf[j] \neq \emptyset \wedge ctr \leq k$	$- - -$
pass-tk	$MEDIUM = \emptyset \wedge (outbuf[j] = \emptyset \vee ctr = k + 1)$	$MEDIUM \leftarrow (T, next, i, \emptyset)$

Table 1: Predicate-Action Table for the Network Nodes

the initial value. The symbols "$\vee$" and "$\wedge$" indicate the logical OR and logical AND operations, respectively. The notation $MEDIUM.(t, DA)$ is used to denote the first two fields of the variable *MEDIUM*. For example, $MEDIUM.(t, DA) = (T, i)$ is a boolean expression which is true if and only if the first field of *MEDIUM* contains the value T, and the second field contains the value i (see the *get-tk* transition).

Some observations concerning this simplified specification are in order. As previously mentioned, the channel is assumed to be error free. This means that the clearing of the medium by the receiver may be taken as an acknowledgement by the sender. There is thus no need for error checking on the channel (such as the Frame Check Sequence); this field was left out of the initial specification. This also means there is no need for timers and timeouts. In Chapter 6, we show how to relax some of these assumptions. However, this specification does contain the main idea of the token bus protocol, and analysis for the logical behavior of the machines can be performed. The following chapter contains this analysis.

IV. ANALYSIS OF THE PROTOCOL

There are at least two major types of analysis which are carried out on communication protocols; one is commonly referred to as *performance analysis*, the other as *formal modeling* of protocols. In performance analysis, the protocol is given, and some assumptions are made concerning the "inputs" to the protocol system – for instance, the mean and probability distribution of the arrival of packets – and the task is to determine how well the protocol performs, in some sense. One example is to determine the maximum throughput of the protocol. The modeling tools which are used are generally taken from probability and queueing theory.

The formal modeling of protocols is concerned with the design of the protocol, its proper specification, with its analysis for freedom from errors and functional correctness, and with implementation and testing. The analysis tends to be exact rather than probabilistic, so the modeling tools used in this analysis are similar to those used in the analysis of algorithms in computer science. Some examples are finite state machines, Petri nets, and programming language models.

The analysis in this thesis is of the second type. From the formal specification of the previous section, certain safety and liveness properties concerning the token bus protocol are derived.

One of the methods of analysis with this protocol model, *systems of communicating machines*, is called *system state analysis* [12]. In Figure 4, the system state analysis of the token bus protocol is given for two machines. The two element tuple, (0,0), in the upper left hand corner of Figure 4 is the initial system state. It indicates that station 1 (the left element) is in state 0, as is station 2 (the right element). The other initial condition is that *MEDIUM* contains a token message with highest numbered station in

the network being the *DA*. Notice that the only path from (0,0) is a *get-tk* transition. When this occurs, station 2 moves to state 2 and the system moves to system state (0,2). The methodology of this *system state analysis* then is to continue to transition, in accordance with Table 1, on each of the out arcs as they are reached, utilizing a separate finite state machine for each station and a global entity as the *MEDIUM*. The result of each of these transitions will be a new system state tuple. The idea is to ensure that every system state is reachable and that there are no system states in which a station or the system can not transition out of. The analysis of Figure 4 does show that for a network of two machines, the protocol is free from deadlocks and nonexecutable transitions.

The analysis of Figure 5 shows that for a network of three machines the protocol is also free from deadlocks and nonexecutable transitions. Note that the additional complexity is minimal when incrementing the number of machines. Although not included here, thesis workups included *system state analysis* of networks of up to ten machines.

However, in order to analyze an arbitrary number of machines, a more general proof is necessary. The following proofs are a generalization of the *system state analysis* to an arbitrary number of machines.

First, it is shown that the protocol possesses the most basic safety property, freedom from deadlocks. In order to do so, we must show that the network must always continue to move from one state to the next, for all possible reachable states. We show first that the token will be passed indefinitely by non-transmitting stations (Lemma 1). Then it is shown that a station with data will also pass the token (Lemma 2). Then these are combined to show freedom from deadlocks, in Theorem 1. Lemma 3 gives the number of states in a system state analysis for the protocol.

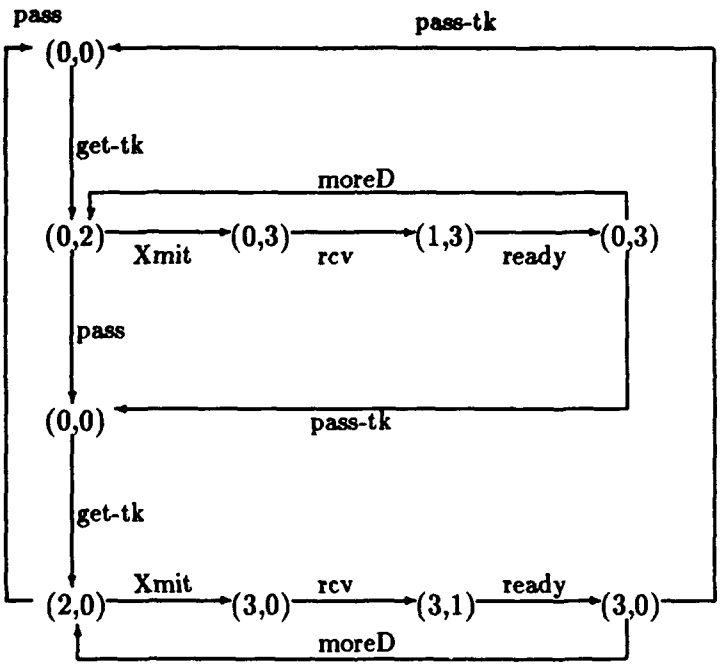

Figure 4: System State Analysis: Two Machine Network

Lemma 1 *From the initial system state, if no station has data messages to transmit, the token will be passed from station to station, returning to the initial system state in exactly $2n$ transitions.*

proof. Initially each machine is in state 0, and the shared variable $MEDIUM$ contains the token addressed to some station i; that is,

$$MEDIUM.(t, DA) = (T, i).$$

The *get-tk* transition will thus be enabled in station i, and no other transition in any other machine will be enabled; so this transition will be taken, moving station i to state 2. Since station i has no messages to send, we have $inbuf[j] = \emptyset$; so the next transition to occur is the *pass*, returning station i to state 0, and placing the token into $MEDIUM$ with the next station, the address of which is in local variable *next*, as the destination.

An identical sequence of events will then occur in the next station, and the next, until the token returns to station i. Since there are exactly n stations on the network, and each station executes exactly two transitions, a total of $2n$ transitions are executed before the token returns to station i. □

Lemma 2 *If any station with $1 \leq m \leq k$ messages to transmit acquires the token, this station will transmit all* m *messages and pass the token on to the next station on the logical ring.*

proof. Assume that station i is in state 2 (having acquired the token) and all other stations are in state 0. Since the station has at least one message to transmit, we have $outbuf[j] \neq \emptyset$. Thus the *Xmit* transition is enabled, and no other transitions are enabled, so station i must move to state 3, while all other stations remain in state 0. This action writes the contents of the input buffer into $MEDIUM$, with some station, say l, as the destination. From state 3, sending station i now has no action enabled (since $MEDIUM \neq \emptyset$). The station to whom the message was addressed, station l, may now take the *rcv* transition to state 1. Next, station l takes the *ready* transition back to state 0, which clears $MEDIUM$, enabling station i to take either the *pass-tk* or *more-D* transition. Either $outbuf[j] = \emptyset$ is true or false. (Observe that j was incremented by the *Xmit* action). If true, the *pass-tk* transition is enabled, and the station writes the token into $MEDIUM$, completing the proof.

If $outbuf[j] \neq \emptyset$, the *more-D* transition is taken, and machine i returns to state 2. From this state, the same sequence of transitions will be taken, which transfers the next data block to its receiver. This sequence will be repeated until all data blocks are transmitted, indicated when *ctr* reaches the value k, or when the next buffer is empty, indicated by $outbuf[j] = \emptyset$, at which time the *pass-tk* transition will be taken. □

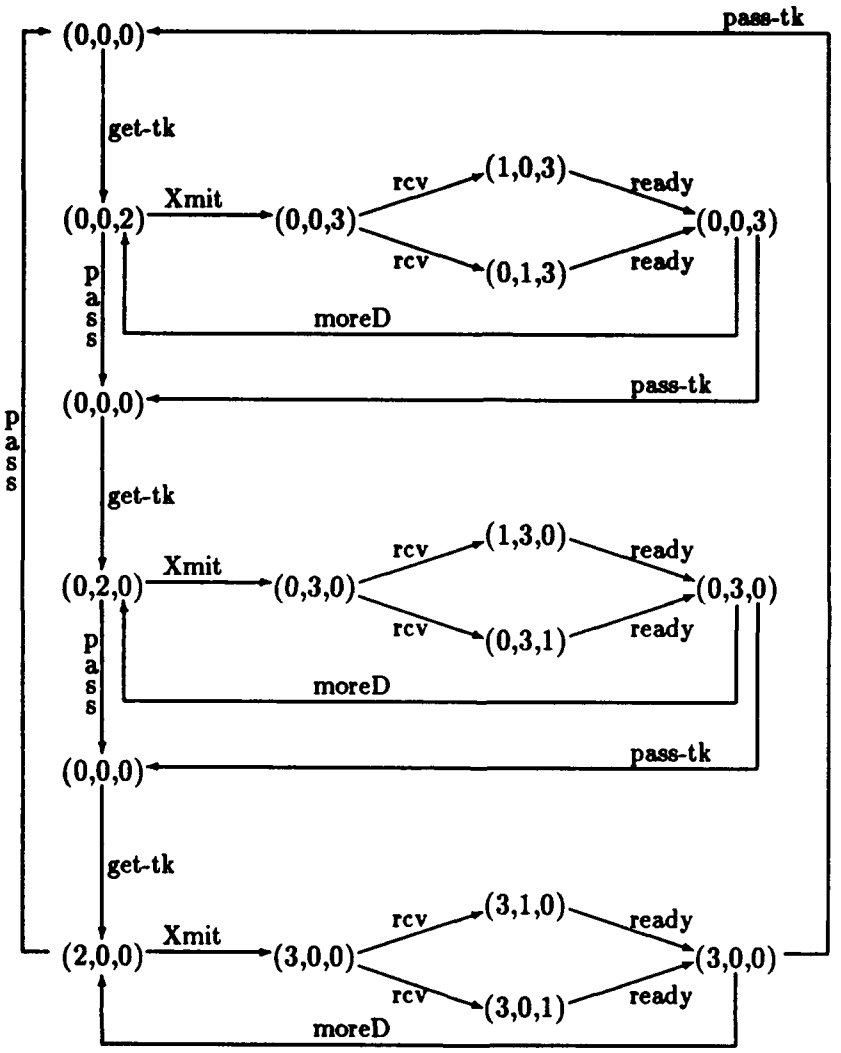

Figure 5: Extended System State Analysis: Three Machine Network

Lemma 3 *The system state reachability analysis for the token bus protocol as specified has $n(n+3)$ system states.*

proof. One complete pass of the token around the logical ring generates $2n$ states. For exactly n of these, one station is in possession of the token, and may transmit. From each of these n states, the *Xmit* transition leads to one more state. This station may transmit to any of the other $(n-1)$ stations, which then receives the data frame. This receiving transition adds $(n-1)$ more states. The *ready* transition then leads to one more state. Thus, for each of the n transmit states, there are $1+(n-1)+1=(n+1)$ states. Thus there are a total of $2n+n(n+1)=n(n+3)$ system states. □

Note the three machine network ($n=3$) of Figure 5. A simple count of the system states shows that this lemma holds. When $n=3$, $n(n+3)=18$. Returning to Figure 4, it can be seen that the property holds for the two machine network as well. In fact the property holds for an arbitrary number of machines.

Theorem 1 *(Safety) The token bus protocol as specified is free from deadlocks.*

proof. It suffices to show that if the network is in the initial system state,

(0,0,...,0)

then it will eventually leave this state and return to this state in a finite number of transitions.

Without loss of generality, assume that there are n stations on the network, that their addresses are 1,2,...,n, and that the value of *next* for station i is $i-1$, excepting station 1, for which $next = n$.

In the initial state, the value of the shared variable *MEDIUM* is $(T, i, p, \emptyset)$ for some network node i. Since i is in state 0, the *get-tk* is enabled, and by fairness will eventually occur; thus the system has left the initial system state.

If station i has no data to send, it will pass the token on to the next station, by Lemma 1; and, similarly, each station without data will pass the token, until the first station with data to send, say station l, receives the token.

Station l, having data to send, will send the data and then pass on the token to the next station, by Lemma 2.

Next, station l's downstream neighbor (the address in local variable *next* of l) will receive the token and (1) pass it on to the next station, if it has no data to send (Lemma 1), or (2) send the data and then pass the token (Lemma 2).

Thus the token will be passed on from one station to the next, and eventually returns to station i, at which point the system has returned to its initial state. □

Corollary 1 *The token bus protocol as specified is free from nonexecutable transitions.*

The proof of the corollary is contained in the proofs of Lemmas 1 and 2, and Theorem 1. The reader may verify this by listing each transition which is a part of the protocol specification, and noting that at some point in the proofs each transition is enabled, and may thus be executed. □

Freedom from deadlocks is the most basic safety property. A deadlock occurs when all machines in the system reach a state in which no further progress is possible. A nonexecutable transition does not necessarily lead to an error in the execution of the protocol; it is simply a transition which can never be executed. However, since transitions are put into a specification for some purpose, the existence of a nonexecutable transition may be considered to be a design error.

Liveness is another important property. Liveness in a network, in contrast to not being deadlocked, means that real progress is being made as transitions occur and tokens and messages propogate throughout the network. Not being deadlocked is one thing, but being *live* is considerably different. The next theorem proves liveness in the specified token bus protocol.

Theorem 2 *(Liveness) For a network of* $2 \leq n$ *stations, any message in the variable* outbuf *of station* i *which has* j *as the destination address (DA),* $i \neq j$ *and* $1 \leq i, j \leq n$, *will eventually appear in the variable* inbuf *of station* j.

proof. Suppose that station i has a message to send to station $l, i \neq l$. Then the DA field of $outbuf[j]$ has the value l. Eventually by Theorem 1, i will get the token, passing to state 2. From state 2, the transition $Xmit$ is enabled, leading station i to state 3, and copying the contents of $outbuf[j]$ into $MEDIUM$. In state 3, station i is now blocked.

Station l, however, now has the *rcv* transition enabled as $MEDIUM.DA = l$. Taking this transition, the value of $MEDIUM$ is copied into the local variable $inbuf$ of station l. □

The proof of the liveness property means that any station on the network with data to send to another station will eventually acquire the token and successfully transmit the data to its receiver. Freedom from deadlocks means that the network will not halt.

V. A WORKING PROGRAM

A desire to demonstrate the completeness of the Chapter 3 specification and the accuracy of the Chapter 4 analysis led to the writing of a program, "tkbus.C", that is included as APPENDIX A to this thesis. It was written in an object-oriented language, C++, and is the driver for a simplified token bus network of three independent stations. More stations could easily be added to the program, but it was determined that the functionality required for a demonstration was adequately provided by three stations. During the implementation, as many as six stations were tested, but having more than three stations proved to introduce no new problems other than bulk and clutter. Thus, the program in Appendix A drives three stations in a simulated token bus network as specified in Chapter 3.

Each independent station is modeled as an independent finite state machine object. A separate C++ header file sets up the structures and executes the transitions for the four-state finite state machine seen in Figure 3. Note that each header file constructs a separate four-state finite state machine object. The self-explanatory names of these included header files are "tkbus-sta1.h", "tkbus-sta2.h", and "tkbus-sta3.h".

First, within each station header file, external links are made to the shared fields of *MEDIUM* which are defined and declared in the main program, "tkbus.C". Then a parent structure is set up to provide the variables local to the four states of this machine only. These local variables can also be seen in Figure 3. Next, member functions provide all the outgoing arc pointers. These pointers correspond to the paths that are followed when a transition moves a machine from one state to another.

The functionality of each finite state machine, as specified in the Predicate-Action Table for Network Nodes (Table 1), is then provided by the last four member functions in the file. Comments, such as, /* *get-tk* */ and /* *Xmit* */ , are positioned to highlight

where the transition logic resides. Note the predicates which must be met and the actions to be taken when the predicates are true. All station's header files were commented identically for conciseness. The comments should provide assistance to readers unfamiliar with C or C++ code.

Having set the stage with three autonomous machines, all initialized to state 0, main() of "tkbus.C" is now set to drive the network. Other initialization values show the token holding time (THT) is set to a predetermined maximum number of messages that a station can transmit each time it possesses the token, and *MEDIUM* contains a token message addressed to the highest numbered station in the network; in this case, $MEDIUM.DA = 3$. The initial transition is now taken. This is *get-tk* at station 3. Because $MEDIUM.t = \mathrm{T}$ and $MEDIUM.DA = 3$, station 3's machine has the one and only enabled predicate. Therefore, it is the only station that can transition. Now that station 3 has transitioned to state 2 and it has the token, it can *Xmit* a message. This is done by writing a message from *outbuf* into the fields of the shared variable, *MEDIUM*. Station 3 has now transitioned to state 3. Note that only station 3 has satisfied any predicates and taken action up to this point. But now station 3 is prevented from continuing. Only the station that corresponds to the value written in *MEDIUM.DA* can transition. This station will meet the predicate for a *rcv* transition to state 1, immediately become *ready*, clear the *MEDIUM*, and transition back to state 0. Now that the *MEDIUM* is clear, station 3 can continue to transmit up to the maximum THT or *pass-tk*; whatever the case may be. All station's machines transition in the same manner. The token passes to all stations and the above sequence, with different transmitting stations and different receiving stations, will continue until the program is terminated.

Not discounting the THT, which is a limiting factor that keeps the token moving from station to station, it is important to note that the driving factors of the specification, the program, and the protocol in general, are quite simple. They are; the presence or absence of a value in the *MEDIUM* fields, or the presence or absence of a message in

each station's *outbuf* when it has the token. All Table 1 predicates, (if statements in the program), are based on this simple observation.

When $outbuf = \emptyset$ at all stations, (all station's outbufs are empty), the program's behaviour reinforces the statements of Lemma 1. A trace of this execution is included as APPENDIX B. When the outbufs of various stations are set to various values that indicate different amounts of messages available for transmission at different stations, (a realistic network situation), the program's behaviour reinforces the statements of Lemma 2, Theorem 1, Theorem 2, and Corollary 1. A trace of this realistic execution of the "tkbus.C" program is included as APPENDIX C. It is easy to see from this trace, all of the transitions that occurred and the order in which they occurred. (All transitions start on the left margin). Also shown is the contents of the *MEDIUM* throughout the execution.

In summary, the program "tkbus.C" and its included header files do reinforce the validity of Chapter 3 and Chapter 4 of this thesis. A similar program could be used to simulate and test a larger subset, or even the entire token bus protocol, as specified in the IEEE 802.4 Standard.

VI. EXTENSIONS OF THE PROTOCOL

In the specification of Chapter 3, several simplifying assumptions were made. The most critical of these was that the communication channel, represented by the shared variable *MEDIUM*, was error free. As a result, no provision was made for acknowledgments or timeouts. In this chapter we show how to remove some of these restrictions.

The token bus protocol allows only one station to access the channel, the machine in possession of the token. Strict adherence to this rule means that a station receiving a message is unable to access the channel for the purpose of sending an acknowledgment, until the token has been passed to it. This means that the sending station must give up the token in order to learn whether its message was received; then, if the message was not received, the sender must again wait until receiving the token before retransmitting the message.

The solution to this problem is simple. After sending a message, the sender allows the receiver to access the channel for the explicit purpose of acknowledgment only, before passing the token or sending any further messages. This is accomplished in the 802.4 Standard by having the token holder transmit a "request with response" data frame. This type of data frame signals a receiving station to immediately respond with an acknowledgment upon receipt of the message. If no acknowledgment is received within a specified time, the sender assumes that the message was not properly received and retransmits that message.

This acknowledgment provision has been added, and the resulting specification is shown in Figure 6. One new state and four new transitions are presented, along with modified versions of the shared variable $MEDIUM$ and the local variables *inbuf* and *outbuf*. The FC (frame control) field of these variables is an expanded specification of

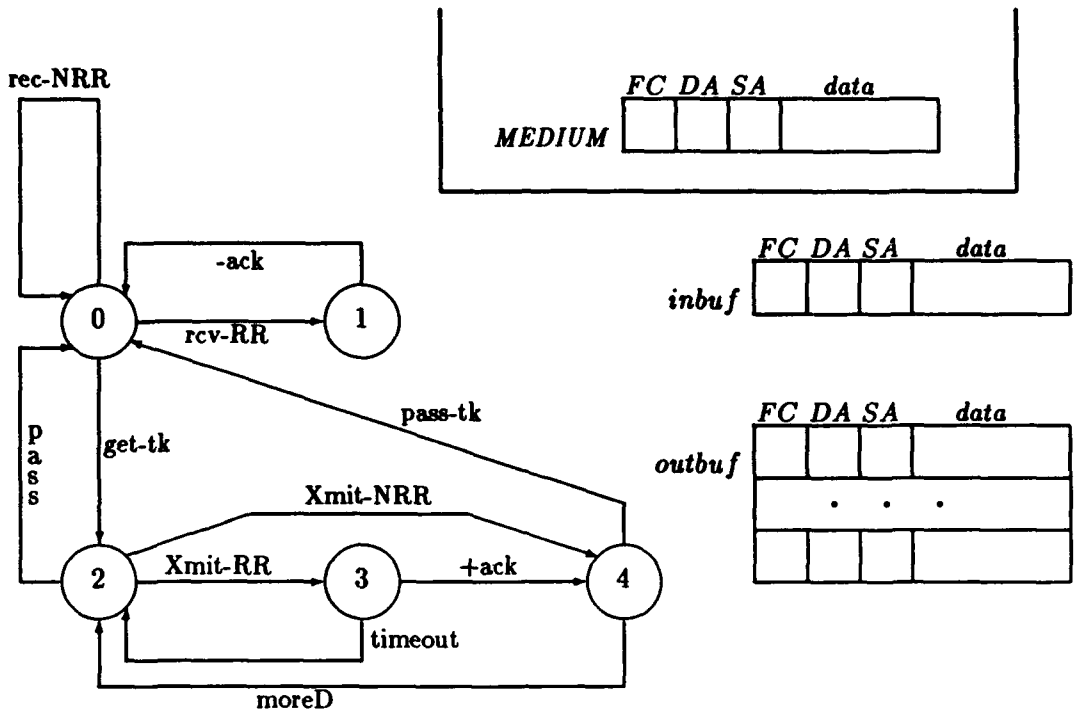

Figure 6: Extended Specification

transition	*predicate*	*action*
get-tk	$MEDIUM.(FC, DA) = (T, i)$	$MEDIUM \leftarrow \emptyset$; $ctr \leftarrow 1$
pass	$outbuf[j] = \emptyset \ \vee \ ctr = \text{max+1}$	$MEDIUM \leftarrow (T, next, i, \emptyset)$
Xmit-NRR	$outbuf[j] \neq \emptyset \ \wedge \ ctr \leq \text{max} \ \wedge$ $outbuf[j].FC = NRR$	$MEDIUM \leftarrow outbuf[j]$; $ctr \leftarrow ctr + 1$; $j \leftarrow j \oplus 1$
Xmit-RR	$outbuf[j] \neq \emptyset \ \wedge \ ctr \leq \text{max} \ \wedge$ $outbuf[j].FC = RR$	$MEDIUM \leftarrow outbuf[j]$; *set-timer*; $ctr \leftarrow ctr + 1$
rcv-NRR	$MEDIUM.(FC, DA) = (NRR, i)$	$inbuf \leftarrow MEDIUM$
rcv-RR	$MEDIUM.(FC, DA) = (RR, i)$	$inbuf \leftarrow MEDIUM$
−ack	$TRUE$	$MEDIUM \leftarrow (A, inbuf.SA, i, \emptyset)$
+ack	$MEDIUM.(FC, DA) = (A, i)$	$MEDIUM \leftarrow \emptyset$; $j \leftarrow j \oplus 1$
moreD	$outbuf[j] \neq \emptyset \ \wedge \ ctr \leq \text{max}$	− − −
pass-tk	$outbuf[j] = \emptyset \ \vee \ ctr = \text{max+1}$	$MEDIUM \leftarrow (T, next, i, \emptyset)$
timeout	(timer expires)	$MEDIUM \leftarrow \emptyset$

Table 2: Revised Predicate-Action Table

the earlier t field. It indicates what type of message is on the $MEDIUM$ or what type of message is buffered. The *Xmit* and *rcv* transitions have been split to provide for transmitting data frames with no response required. *Xmit-NRR* covers transmitting with no response required. For transmitting data frames that require a response, the *Xmit-RR* transition is taken. This is similar to the previous *Xmit* transition, however, the action of *Xmit-RR* includes setting the timeout timer. The *rcv* transition has been split into the *rcv-NRR* for no response required, and for receiving frames where a response is required the *rcv-RR* transition is used. The FC field of $MEDIUM$ and both buffers takes *NRR* and *RR* as values.

The *ready* transition has been modified and renamed *-ack* (send acknowledgment). It is enabled only in state 1, which is reached only when the FC field of the transmitted message indicates the sender of the message requests a response. When this is true, the receiver causes $MEDIUM$ to become an acknowledgment message and $MEDIUM.FC$ becomes A. The fourth value that FC can take on is T for token, (this is identical to the previous specification).

The last modification is strictly a cosmetic one. The word *max* replaces the letter k from Chapter 3. It is more inherently descriptive and indicates the maximum number of messages that can be transmitted during a single token holding time.

Summarizing the changes, upon taking the *Xmit-RR* transition, the sending machine sets a timer. In state 3, the sender waits until receiving either the acknowledgment, or a timeout. Because the FC field of the data frame indicates that a response is required, the receiving station will transition to state 1 and copy $MEDIUM$ into its *inbuf*. An acknowledgment data frame is then placed on the *MEDIUM* by the receiver and the transition back to state 0 is taken. Upon receiving this acknowledgment, the sender transitions to state 4. (This is equivalent to state 3 of the previous specification). If a timeout is received, the transmitting station returns to state 2 and resends the message. If an outgoing data frame does not require a response, the sender's *Xmit-NRR* transition

will be taken. Since no response is required this station will immediately transition to state 4. The receiving station will interpret no response required, take the *rec-NRR* transition, and loop back into state 0. All changes to transition predicates and actions are presented in Table 2.

The system state reachability analysis for this extension has been carried out, and has $n(n+4)$ system states. This is derived in the same manner as Lemma 3 of the previous specification but provides for the additional state that must be implemented in each machine. In lieu of a formal proof, it is sufficient to discuss this in the following way. There are five transitions for each of the n machines that result in a single system state. These are *get-tk, pass, Xmit, -ack,* and *+ack.* Therefore, in a network of n stations there will be $5n$ states resulting from these transitions. All that remains to quantify is the *rcv* transition which splits to $n-1$ receiving stations. In a network of n stations then, there will be $n(n-1)$ transitions resulting from these *rcv* transitions. Combining the terms results in the following:

$$5n + n(n-1) = n(5+n-1) = n(n+4)$$

As with the previous specification, this property will hold for an arbitrary number of machines. An extended system state analysis for a three station network is shown in Figure 7. Note that there are $n(n+4) = 21$ system states. The analysis shows that this extended protocol is also free from deadlocks.

In order to cause errors to occur in the communication channel, another machine, called *demon* could be added to the network. This machine would only have one state and one transition. The single transition would clear the *MEDIUM* to "empty" from time to time.

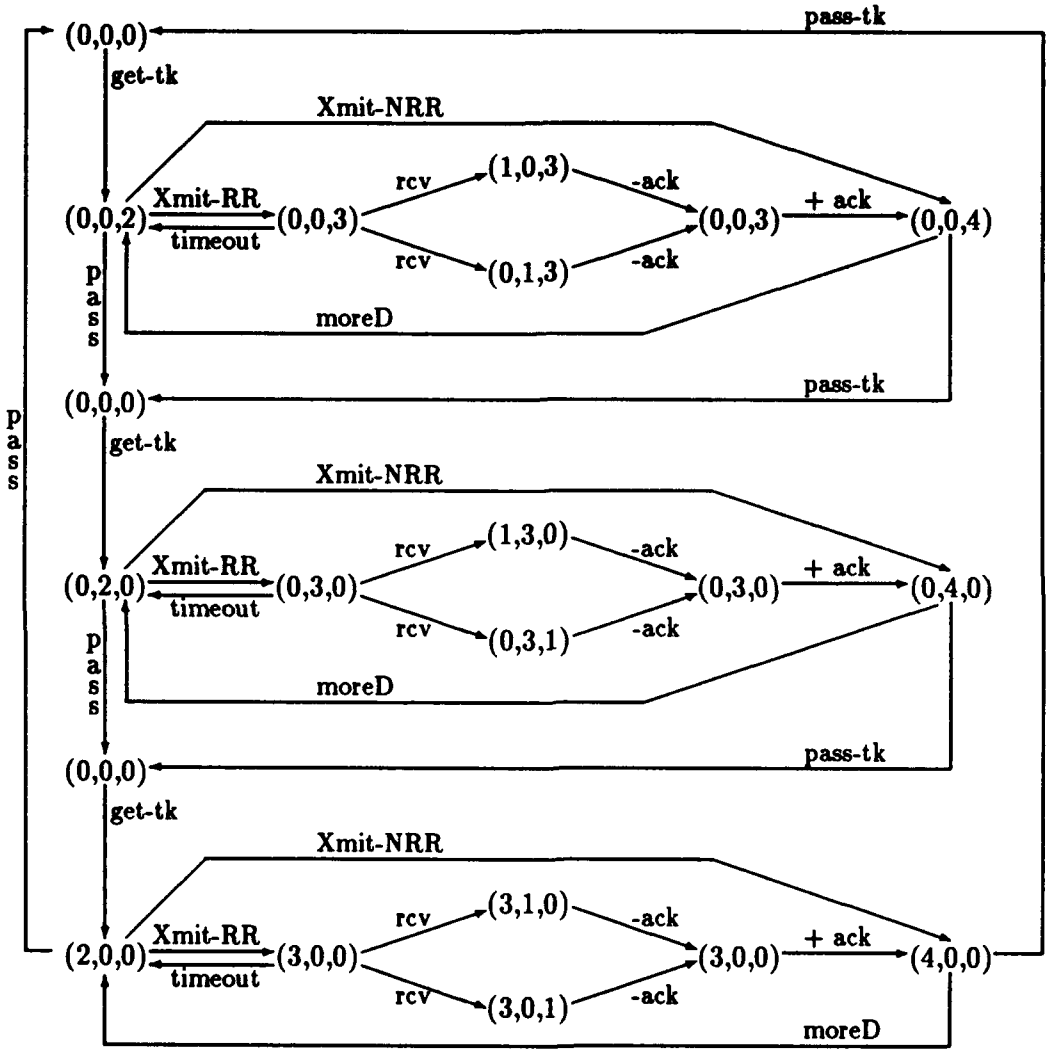

Figure 7: Extended System State Analysis: Three Machine Network

That is, let the transition name be *delete*. The enabling predicate for *delete* is

$$MEDIUM.FC = RR \quad \vee \quad MEDIUM.FC = NRR$$

and the associated action is

$$MEDIUM \leftarrow \emptyset.$$

Because the *delete* transition of the *demon* machine and at least one of the *rcv* transitions of the receiving machine will be enabled at the same time, any data message written onto the channel has the possibility of being lost. Note that this enabling predicate only allows data messages to be deleted (not tokens or acknowledgements). Lost tokens and acknowledgements can be handled in a similar manner, but are not included here for the sake of brevity.

A word should also be said concerning the timer that magically enables the *timeout* transition. This timer and the concept of time have not been formally defined here. However, it is simply assumed that a machine needing a timer can set one, and be interrupted when the timer expires. Of course this is frequently done by programs in computers, but for a formal analysis the timer should be included as a part of the formal specification and definition. One possible way of doing this is to include a timer as an additional, subordinate machine to the network station. This is done, in fact, in other research on this model for a high speed protocol, FDDI (fiber distributed data interface) [14].

VII. SUMMARY

The introduction of this thesis described the importance of formally specifying and analyzing communication protocols. Early methods that pre-date the model used here were mentioned. Then a short description of some well known protocols, their history, and analysis completed on them were presented. The topic of this thesis, a formal specification and analysis using *systems of communicating machines* on a token bus protocol was then introduced.

A formal specification of the token bus protocol was given, as well as an analysis for safety and liveness properties. The analysis showed that the protocol was free from deadlocks and nonexecutable transitions, and that progress in communication must also occur.

The method used to specify the protocol was a formal description technique, or model, called *systems of communicating machines* [12-18]. This is a model designed especially for computer communication networks, which uses a combination of finite state machines and variables in the specification of each machine, and shared variables for communication between machines. An analysis technique called *system state analysis* was applied to a network of two machines. This method is unique to this protocol model, although the idea is similar to some other methods of reachability analysis. The analysis was then generalized through proofs to an arbitrary number of machines. Proofs showed that the protocol is free from deadlocks and nonexecutable transitions, and that the successful transfer of data is guaranteed.

Following the proofs, a working program written in the object oriented language, C++, was presented. This program treated each machine of a three station network as an independent object. It demonstrated that the specification and analysis of the protocol, as presented in the thesis, were complete and accurate.

Finally, the protocol specification was extended to include more features of the token bus as specified in the IEEE Standard. This extension included transmitting and receiving two types of data frames, acknowledgments sent from the receiver, timeouts in the sender, and receipt of acknowledgments at the sender. An extended *system state analysis* for a network of three machines was then provided.

Transmission errors (losses) in the channel were modeled through the use of an additional machine called *demon*, which arbitrarily deleted data messages appearing in the channel. Success of the *demon* would lead to timeouts in the network machines.

This thesis has shown the applicability of *systems of communicating machines* to the modeling of a well-known protocol, and has shown the logical behavior and strengths of that protocol. It provides confirmation that the model is useful for formal specification and should be considered a viable technique for the development of industrial standards for communication protocols and other complex software.

APPENDIX A

C++ CODE

```
//  Modeling the Token Bus Protocol with Systems of Communicating Machines.
//
//  Author:  L. J. Charbonneau, LT, USN                May 1990
//  System:  Vax 11/780
//  Compiler: C++ Version 1.2
//
//     This file is a C++ header file that sets up the structure and
//  executes the transitions for a four state finite state machine object.
//  The independent object is station 1 of a simplified token bus network.
//  This stations's ID is 1.  The outbuf for this station
//  is set at 5. This means that 5 messages are available for transmission
//  every time station 1 gets the token.



// File tkbus_sta1.h

#include <string.h>

// The following fields are defined externally in tkbus.C and are visible
// at all times at all station objects.
extern int n;              /* n = number of nodes in the network.*/
extern THT;                /* THT = max token holding time of one station.*/
extern int medium;         /* 1 = comm_bus is busy. 0 = comm_bus is empty. */
extern char t;             /* t = type of frame; T = token, D = data */
extern int da;             /* da = dest address of message.  Who gets it. */
extern int sa;             /* sa = source address of message. Who sent it. */
extern char* data_msg;     /* contents of the messages on the network. */
```

```
struct parent1          /* Parent structure for the 4 states in this FSM  */
{
  static char *data;
  static int  i;
  static int  next;
  static int  ctr;
  static int  inbuf;
  static int  outbuf;
  static int  doom_state;

  parent1()
  {
    data = "data...msg...from...sta1...";
    i = 1;                          /* Station ID = 1. */
    next = n;                       /* Logical next = station n.*/
    ctr = 0;
    inbuf = 5;
    outbuf = 0;
    doom_state = 0;
  }

  virtual parent1 *transition() {};
};
```

```
/* Structure for state 0. */

struct n1state0 : public parent1
{
   parent1 *ptr1, *ptr2;    /* state 0 has out arcs to state 1 and state 2.*/
   n1state0() : () {}
   parent1 *transition();
};



/* Structure for state 1. */

struct n1state1 : public parent1
{
   parent1 *ptr0;           /* state 1 has an out arc only to state 0.*/
   n1state1() : () {}
   parent1 *transition();
};



/* Structure for state 2. */

struct n1state2 : public parent1
{
   parent1 *ptr0, *ptr3;    /* state 2 has out arcs to state 0 and state 3.*/
   n1state2() : () {}
   parent1 *transition();
};



/* Structure for state 3. */

struct n1state3 : public parent1
{
   parent1 *ptr0, *ptr2;    /* state 3 has out arcs to state 0 and state 2 .*/
   n1state3() : () {}
   parent1 *transition();
};
```

```
/* This function performs the transitions when current state = state 0.    */

parent1 *n1state0::transition()
{
                                        /* If it's a token for you, */
  if ((t == 'T') && (da == i))
  /* get_tk */      { medium = 0;       /* Clear the medium.       */
                    t = ' ';
                    da = 0;
                    sa = 0;
                    ctr = 1;            /* Set message counter to 1.*/
                    printf("\n\nget_tk_sta1");
                    return ptr2; }      /* GO TO STATE2.           */

                                        /* If it's a msg for you,  */
  if ((t == 'D') && (da == i))
  /* rcv */         { inbuf = medium;   /* Copy it to your inbuf.  */
                    printf("\nrcv_msg_at_sta1");
                    return ptr1; }      /* GO TO STATE1.           */
    else            doom_state = 1; return this;
}


/* This function performs the transitions when current state = state 1.    */

parent1 *n1state1::transition()
{
                  medium = 0;           /* Clear the medium.       */
  /* ready */     t = ' ';
                  da = 0;
                  sa = 0;
                  printf("\nsta1_ready");
                  printf("\n  medium = .....clear");
                  return ptr0;          /* GO TO STATE0.           */
}
```

```
/* This function performs the transitions when current state = state 2.    */

parent1 *n1state2::transition()
{
  if (outbuf == 0)                          /* If outbuf is empty,     */
  /* pass */        { t = 'T';              /*  send token msg to next.*/
                    da = next;
                    sa = i;
                    printf("\npass. No_msgs_in_outbuf_at_sta1");
                    printf("\n  medium = tk...msg...for...sta%d",da);
                    return ptr0; }          /* GO TO STATE0.          */

  if (outbuf != 0)                          /* If outbuf not empty,    */
  /* Xmit */        { medium = 1;           /*  put data msg on medium.*/
                    t = 'D';
                    da = 2;
                    sa = 1;
                    data_msg = data;
                    printf("\nxmit_msg_from_sta1");
                    printf("\n  medium = %sfor...sta%d",data_msg,da);
                    outbuf--;               /* pt to next msg in outbuf */
                    ctr++;                  /* increment msg counter   */
                    return ptr3; }          /* GO TO STATE3.          */
    else
                  { doom_state = 1; return this; }
}
```

```
/* This function performs the transitions when current state = state 3.    */

parent1 *n1state3::transition()
{
                                        /* If medium is empty AND   */
                                        /*  ( (outbuf is empty) OR  */
                                        /*    (ctr is over THT) ),  */
  if ((medium == 0) && ((outbuf == 0) || (ctr > THT)))
  /* pass_tk */     {  t = 'T';              /*  send token msg to next. */
                    da = next;
                    sa = i;
                    printf("\npass_tk_from_sta1");
                    if (ctr > THT)
                      { printf("\n  THT...exceeded"); }
                    else if (outbuf == 0)
                      { printf("\n  outbuf...is...empty"); }
                    printf("\n  medium = tk...msg...for...sta%d",da);
                    outbuf = 5;
                    ctr = 0;
                    return ptr0;  }          /* GO TO STATE0.          */

                                        /* If medium is empty AND   */
                                        /* outbuf is not empty AND  */
                                        /* ctr is <= THT ,          */
  if ((medium == 0) && (outbuf != 0) && (ctr <= THT))
  /* more_D */      {
                    printf("\nmore_data_in_outbuf_at_sta1");
                    return ptr2;  }          /* GO BACK TO STATE2.     */
     else           doom_state = 1; return this;
}
```

```
//  Modeling the Token Bus Protocol with Systems of Communicating Machines.
//
//  Author:  L. J. Charbonneau, LT, USN                May 1990
//  System:  Vax 11/780
//  Compiler: C++ Version 1.2
//
//     This file is a C++ header file that sets up the structure and
//  executes the transitions for a four state finite state machine object.
//  The independent object is station 2 of a simplified token bus network.
//  This stations's ID is 2.  The outbuf for this station
//  is set at 1. This means that 1 message is available for transmission
//  every time station 2 gets the token.


// File tkbus_sta2.h

#include <string.h>

// The following fields are defined externally in tkbus.C and are visible
// at all times at all station objects.
extern int n;              /* n = number of nodes in the network.          */
extern THT;                /* THT = max token holding time of one station. */
extern int medium;         /* 1 = comm_bus is busy. 0 = comm_bus is empty. */
extern char fc;            /* fc = type of frame; t = token, d = data.     */
extern int da;             /* da = dest address of message.  Who gets it.  */
extern int sa;             /* sa = source address of message. Who sent it. */
extern char* data_msg;     /* contents of the messages on the network.     */
```

```
struct parent2          /* Parent structure for the 4 states in this FSM  */
{
   static char *data;
   static int  i;
   static int  next;
   static int  ctr;
   static int  inbuf;
   static int  outbuf;
   static int  doom_state;

   parent2()
   {
      data = "data...msg...from...sta2...";
      i = 2;                            /* Station ID = 2.        */
      next = i - 1;                     /* Logical next = station 1.*/
      ctr = 0;
      inbuf = 1;
      outbuf = 0;
      doom_state = 0;
   }

   virtual parent2 *transition() {};
};
```

```
/* Structure for state 0. */

struct n2state0 : public parent2
{
   parent2 *ptr1, *ptr2;    /* state 0 has out arcs to state 1 and state 2 */
   n2state0() : () {}
   parent2 *transition();
};



/* Structure for state 1. */

struct n2state1 : public parent2
{
   parent2 *ptr0;           /* state 1 has an out arc only to state 0     */
   n2state1() : () {}
   parent2 *transition();
};



/* Structure for state 2. */

struct n2state2 : public parent2
{
   parent2 *ptr0, *ptr3;    /* state 2 has out arcs to state 0 and state 3 */
   n2state2() : () {}
   parent2 *transition();
};



/* Structure for state 3. */

struct n2state3 : public parent2
{
   parent2 *ptr0, *ptr2;    /* state 3 has out arcs to state 0 and state 2 */
   n2state3() : () {}
   parent2 *transition();
};
```

```
/* This function performs the transitions when current state = state 0.    */

parent2 *n2state0::transition()
{
                                        /* If it's a token for you, */
  if ((t == 'T') && (da == i))
  /* get_tk */      { medium = 0;       /* Clear the medium.       */
                    t = ' ';
                    da = 0;
                    sa = 0;
                    ctr = 1;            /* Set message counter to 1.*/
                    printf("\n\nget_tk_sta2");
                    return ptr2; }      /* GO TO STATE2.           */

                                        /* If it's a msg for you,  */
  if ((t == 'D') && (da == i))
  /* rcv */         { inbuf = medium;   /* Copy it to your inbuf.  */
                    printf("\nrcv_msg_at_sta2");
                    return ptr1; }      /* GO TO STATE1.           */
     else           doom_state = 1; return this;
}


/* This function performs the transitions when current state = state 1.    */

parent2 *n2state1::transition()
{
                  medium = 0;           /* Clear the medium.       */
  /* ready */     t = ' ';
                  da = 0;
                  sa = 0;
                  printf("\nsta2_ready");
                  printf("\n  medium = .....clear");
                  return ptr0;          /* GO TO STATE0.           */
}
```

```
/* This function performs the transitions when current state = state 2.    */

parent2 *n2state2::transition()
{
  if (outbuf == 0)                          /* If outbuf is empty,     */
  /* pass */        { t = 'T';              /*  send token msg to next.*/
                    da = next;
                    sa = i;
                    printf("\npass. No_msgs_in_outbuf_at_sta2");
                    printf(" \n medium = tk...msg...for...sta%d", da);
                    return ptr0; }          /* GO TO STATE0.          */

  if (outbuf != 0)                          /* If outbuf not empty,    */
  /* Xmit */        { medium = 1;           /*  put data msg on medium.*/
                    t = 'D';
                    da = 3;
                    sa = 2;
                    data_msg = data;
                    printf("\nxmit_msg_from_sta2");
                    printf("\n  medium = %sfor...sta%d",data_msg,da);
                    outbuf--;               /* pt to next msg in outbuf */
                    ctr++;                  /* increment msg counter   */
                    return ptr3; }          /* GO TO STATE3.          */
    else
                  { doom_state = 1; return this; }
}
```

```
/* This function performs the transitions when current state = state 3.    */

parent2 *n2state3::transition()
{
                                        /* If medium is empty AND   */
                                        /*  ( (outbuf is empty) OR  */
                                        /*    (ctr is over THT) ),  */
  if ((medium == 0) && ((outbuf == 0) || (ctr > THT)))
  /* pass_tk */     {  t = 'T';              /*  send token msg to next. */
                    da = next;
                    sa = i;
                    printf("\npass_tk_from_sta2");
                    if (ctr > THT)
                       { printf("\n  THT...exceeded"); }
                    else if (outbuf == 0)
                       { printf("\n  outbuf...is...empty"); }
                    printf("\n  medium = tk...msg...for...sta%d",da);
                    outbuf = 1;
                    ctr = 0;
                    return ptr0;  }          /* GO TO STATE0.          */

                                        /* If medium is e  pty AND  */
                                        /* outbuf is not empty AND  */
                                        /* ctr is <= THT ,          */
  if ((medium == 0) && (outbuf != 0) && (ctr <= THT))
  /* more_D */      {
                    printf("\nmore_data_in_outbuf_at_sta2");
                    return ptr2;  }          /* GO BACK TO STATE2.     */
     else           doom_state = 1; return this;
}
```

```
//  Modeling the Token Bus Protocol with Systems of Communicating Machines.
//
//  Author:  L. J. Charbonneau, LT, USN                May 1990
//  System:  Vax 11/780
//  Compiler: C++ Version 1.2
//
//     This file is a C++ header file that sets up the structure and
//  executes the transitions for a four state finite state machine object.
//  The independent object is station 3 of a simplified token bus network.
//  This stations's ID is 3.  The outbuf for this station
//  is set at 3. This means that 3 messages are available for transmission
//  every time station 3 gets the token.

// File tkbus_sta3.h



#include <string.h>

// The following fields are defined externally in token_bus.C and are visible
// at all times at all station objects.
extern int n;              /* n = number of nodes in the network.          */
extern THT;                /* THT = max token holding time of one station.  */
extern int medium;         /* 1 = comm_bus is busy. 0 = comm_bus is empty.  */
extern char fc;            /* fc = type of frame; t = token, d = data.      */
extern int da;             /* da = dest address of message.  Who gets it.   */
extern int sa;             /* sa = source address of message. Who sent it.  */
extern char* data_msg;     /* contents of the messages on the network.      */
```

```
struct parent3          /* Parent structure for the 4 states in this FSM  */
{
   static char *data;
   static int  i;
   static int  next;
   static int  ctr;
   static int  inbuf;
   static int  outbuf;
   static int  doom_state;

   parent3()
   {
      data = "data...msg...from...sta3...";
      i = 3;                          /* Station ID = 3.        */
      next = i - 1;                   /* Logical next = station 2.*/
      ctr = 0;
      inbuf = 3;
      outbuf = 0;
      doom_state = 0;
   }

   virtual parent3 *transition() {};
};
```

```
/* Structure for state 0. */

struct n3state0 : public parent3
{
   parent3 *ptr1, *ptr2;    /* state 0 has out arcs to state 1 and state 2 */
   n3state0() : () {}
   parent3 *transition();
};



/* Structure for state 1. */

struct n3state1 : public parent3
{
   parent3 *ptr0;           /* state 1 has an out arc only to state 0     */
   n3state1() : () {}
   parent3 *transition();
};



/* Structure for state 2. */

struct n3state2 : public parent3
{
   parent3 *ptr0, *ptr3;    /* state 2 has out arcs to state 0 and state 3 */
   n3state2() : () {}
   parent3 *transition();
};



/* Structure for state 3. */

struct n3state3 : public parent3
{
   parent3 *ptr0, *ptr2;    /* state 3 has out arcs to state 0 and state 2 */
   n3state3() : () {}
   parent3 *transition();
};
```

```
/* This function performs the transitions when current state = state 0.    */

parent3 *n3state0::transition()
{
                                        /* If it's a token for you, */
  if ((t == 'T') && (da == i))
  /* get_tk */      { medium = 0;       /* Clear the medium.        */
                    t = ' ';
                    da = 0;
                    sa = 0;
                    ctr = 1;            /* Set message counter to 1.*/
                    printf("\n\nget_tk_sta3");
                    return ptr2; }      /* GO TO STATE2.            */

                                        /* If it's a msg for you,   */
  if ((t == 'D') && (da == i))
  /* rcv */         { inbuf = medium;   /* Copy it to your inbuf.   */
                    printf("\nrcv_msg_at_sta3");
                    return ptr1; }      /* GO TO STATE1.            */
    else            doom_state = 1; return this;
}


/* This function performs the transitions when current state = state 1.    */

parent3 *n3state1::transition()
{
                  medium = 0;           /* Clear the medium.        */
  /* ready */     t = ' ';
                  da = 0;
                  sa = 0;
                  printf("\nsta3_ready");
                  printf("\n  medium = .....clear");
                  return ptr0;          /* GO TO STATE0.            */
}
```

```
/* This function performs the transitions when current state = state 2.    */

parent3 *n3state2::transition()
{
  if (outbuf == 0)                          /* If outbuf is empty,     */
  /* pass */       { t = 'T';               /*  send token msg to next.*/
                     da = next;
                     sa = i;
                     printf("\npass. No_msgs_in_outbuf_at_sta3");
                     printf("\n  medium = tk...msg...for...sta%d",da);
                     return ptr0; }         /* GO TO STATE0.           */

  if (outbuf != 0)                          /* If outbuf not empty,    */
  /* Xmit */       { medium = 1;            /*  put data msg on medium.*/
                     t = 'D';
                     da = 1;
                     sa = 3;
                     data_msg = data;
                     printf("\nxmit_msg_from_sta3");
                     printf("\n  medium = %sfor...sta%d",data_msg,da);
                     outbuf--;              /* pt to next msg in outbuf */
                     ctr++;                 /* increment msg counter   */
                     return ptr3; }         /* GO TO STATE3.           */
     else
                   { doom_state = 1; return this; }
}
```

```
/* This function performs the transitions when current state = state 3.    */

parent3 *n3state3::transition()
{
                                        /* If medium is empty AND   */
                                        /*  ( (outbuf is empty) OR  */
                                        /*    (ctr is over THT) ),  */
  if ((medium == 0) && ((outbuf == 0) || (ctr > THT)))
  /* pass_tk */     {  t = 'T';          /*  send token msg to next. */
                    da = next;
                    sa = i;
                    printf("\npass_tk_from_sta3");
                    if (ctr > THT)
                      { printf("\n  THT...exceeded"); }
                    else if (outbuf == 0)
                      { printf("\n  outbuf...is...empty"); }
                    printf("\n  medium = tk...msg...for...sta%d",da);
                    outbuf = 3; ctr = 0;
                    return ptr0;  }      /* GO TO STATE0.          */

                                        /* If medium is empty AND   */
                                        /* outbuf is not empty AND  */
                                        /* ctr is <= THT ,          */
  if ((medium == 0) && (outbuf != 0) && (ctr <= THT))
  /* more_D */     {
                    printf("\nmore_data_in_outbuf_at_sta3");
                    return ptr2;  }      /* GO BACK TO STATE2.     */
     else           doom_state = 1; return this;
}
```

```
//  Modeling the Token Bus Protocol with Systems of Communicating Machines.
//
//  Author:  L. J. Charbonneau, LT, USN              May 1990
//  System:  Vax 11/780
//  Compiler: C++ Version 1.2
//     This file contains the main program and is the driver for a
//  simulated token bus network of 3 independent stations. The stations
//  are individually set up in the three header files that are #included
//  below. Another station can easily be added to the network by including
//  another "tkbusXXX.h" header file. The appropriate variables and
//  constants must be set in this header file to account for station ID,
//  contents of outbuf, destination address of the downstream neighbor, etc.
//  Also, the states must be added for this new node and main() must be
//  modified for initialization and transition execution. Since the
//  concept is the same for each station, you could add many stations.


// File tkbus.C

#include <stdio.h>
#include "tkbus_sta1.h"  /* Station 1 FSM object.          */
#include "tkbus_sta2.h"  /* Station 2 FSM object.          */
#include "tkbus_sta3.h"  /* Station 3 FSM object.          */


/*GLOBAL VARIABLES; ALL ARE VISIBLE AT ALL NETWORK STATIONS. */
int  n = 3;                 /* n = number of nodes in the network.          */
int  THT = 4;               /* THT = max token holding time of one station. */
int  medium = 1;            /* 1 = comm_bus is busy. 0 = comm_bus is empty. */
char t = 'T';               /* t = type of frame; T = token, D = data.      */
int  da = 3;                /* da = dest address of data_msg or tk_msg.     */
int  sa = 0;                /* sa = source address of data_msg. Who sent it.*/
char *data_msg = "no msg yet"; /* contents of the msgs on the network.      */
```

```
/* Station 1 states. */
 n1state0 node1_s0;
 n1state1 node1_s1;
 n1state2 node1_s2;
 n1state3 node1_s3;

/* Station 2 states. */
 n2state0 node2_s0;
 n2state1 node2_s1;
 n2state2 node2_s2;
 n2state3 node2_s3;

/* Station 3 states. */
 n3state0 node3_s0;
 n3state1 node3_s1;
 n3state2 node3_s2;
 n3state3 node3_s3;

/* This function builds the finite state machines for all the stations.    */
void build_state_machines()
{
   node1_s0.ptr1 = &node1_s1; // Set up out arcs from state 0
   node2_s0.ptr1 = &node2_s1;
   node3_s0.ptr1 = &node3_s1;
   node1_s0.ptr2 = &node1_s2;
   node2_s0.ptr2 = &node2_s2;
   node3_s0.ptr2 = &node3_s2;
   node1_s1.ptr0 = &node1_s0; // Set up out arcs from state 1
   node2_s1.ptr0 = &node2_s0;
   node3_s1.ptr0 = &node3_s0;
   node1_s2.ptr0 = &node1_s0; // Set up out arcs from state 2
   node2_s2.ptr0 = &node2_s0;
   node3_s2.ptr0 = &node3_s0;
   node1_s2.ptr3 = &node1_s3;
   node2_s2.ptr3 = &node2_s3;
   node3_s2.ptr3 = &node3_s3;
   node1_s3.ptr0 = &node1_s0; // Set up out arcs from state 3
   node2_s3.ptr0 = &node2_s0;
   node3_s3.ptr0 = &node3_s0;
   node1_s3.ptr2 = &node1_s2;
   node2_s3.ptr2 = &node2_s2;
   node3_s3.ptr2 = &node3_s2;
}
```

```
/* This is the MAIN function of the token bus program.                */

main ()
{
  printf("\nGOING FOR A RIDE WITH THE TOKEN BUS PROTOCOL!\n");

  build_state_machines();

  parent1 *sta1;    /* Pointer to the current state of Station 1.    */
  parent2 *sta2;    /* Pointer to the current state of Station 2.    */
  parent3 *sta3;    /* Pointer to the current state of Station 3.    */



  /* Print out the initialization values. What is each machines       */
  /* current state? What is the THT set to and what is on the medium. */

  printf("\nInitial conditions are: ");
  printf("\n  All stations are in state0 ");
  printf("\n  Token holding time (THT) is set at %d msgs per station",THT);
  printf("\n  medium = tk...msg...for...sta%d",da);


  /* All stations start in state 0 and execute their initial transition. */

  sta1 = node1_s0.transition();
  sta2 = node2_s0.transition();
  sta3 = node3_s0.transition();




  /* If all stations are doomed after the first transition,           */
  if ( (sta1 -> doom_state == 1) &&
          (sta2 -> doom_state == 1) &&
               (sta3 -> doom_state == 1) )
   {
    printf(" \n  *** Invalid initial transition *** \n\n");
   } /* end if */

  int t_counter = 1;  /* Transition counter. Used to terminate tkbus.C. */
```

```
/* This WHILE LOOP enables the program to act like a token bus network.    */
/* Provided that one transition at one station becomes enabled at any      */
/* iteration of this loop, the program will continue until terminated.     */
/* The token will be passed between all of the stations; and messages      */
/* will be transmitted and received.                                       */

/* As long as one station transitioned, and t_counter<100, keep going.     */
while ( ( (sta1 -> doom_state != 1) ||
          (sta2 -> doom_state != 1) ||
               (sta3 -> doom_state != 1) ) &&
                    (t_counter < 100) )  /* Stop at 100 */
                    /* This clause can be removed and  */
                    /* the program will run forever.       */

 { /* Reset the doom_states at all stations.                */
  sta1 -> doom_state = 0;
  sta2 -> doom_state = 0;
  sta3 -> doom_state = 0;

  /* Transition again.  New state = the old state after transition.  */
  sta1 = sta1 -> transition();
  sta2 = sta2 -> transition();
  sta3 = sta3 -> transition();

  t_counter++;                        /* Increment t_counter.  */

  /* If all stations are doomed after the above transition,  ERROR    */
  if ( (sta1 -> doom_state == 1) &&
            (sta2 -> doom_state == 1) &&
                 (sta3 -> doom_state == 1) )
    {
     printf(" \n  *** Invalid transition, DEADLOCK has occurred ***\n\n");
    } /* end if */

 } /* end while */


 printf("\n \n %d Transitions completed. \n \n", t_counter);

 printf("\n YOUR BUS RIDE IS OVER!!! \n \n");

} /* end main */
```

APPENDIX B

SAMPLE get-tk/pass-tk TRACE

**NOTE: The outbufs of all stations were set to 0 before this trace was produced.

Script started on Tue May 29 00:42:02 1990
nps-cs [[1]] tkbus

GOING FOR A RIDE WITH THE TOKEN BUS PROTOCOL!

Initial conditions are:
All stations are in state0
Token holding time (THT) is set at 4 msgs per station
medium = tk...msg...for...sta3

get_tk_sta3
pass. No_msgs_in_outbuf_at_sta3
medium = tk...msg...for...sta2

get_tk_sta2
pass. No_msgs_in_outbuf_at_sta2
medium = tk...msg...for...sta1

get_tk_sta1
pass. No_msgs_in_outbuf_at_sta1
medium = tk...msg...for...sta3

get_tk_sta3
pass. No_msgs_in_outbuf_at_sta3
medium = tk...msg...for...sta2

get_tk_sta2
pass. No_msgs_in_outbuf_at_sta2
medium = tk...msg...for...sta1

get_tk_sta1
pass. No_msgs_in_outbuf_at_sta1
medium = tk...msg...for...sta3

get_tk_sta3
pass. No_msgs_in_outbuf_at_sta3
medium = tk...msg...for...sta2

get_tk_sta2
pass. No_msgs_in_outbuf_at_sta2
medium = tk...msg...for...sta1

get_tk_sta1
pass. No_msgs_in_outbuf_at_sta1
medium = tk...msg...for...sta3

get_tk_sta3
pass. No_msgs_in_outbuf_at_sta3
medium = tk...msg...for...sta2

get_tk_sta2
pass. No_msgs_in_outbuf_at_sta2
medium = tk...msg...for...sta1

get_tk_sta1
pass. No_msgs_in_outbuf_at_sta1
medium = tk...msg...for...sta3

get_tk_sta3
pass. No_msgs_in_outbuf_at_sta3
medium = tk...msg...for...sta2

get_tk_sta2
pass. No_msgs_in_outbuf_at_sta2
medium = tk...msg...for...sta1

get_tk_sta1
pass. No_msgs_in_outbuf_at_sta1
medium = tk...msg...for...sta3

get_tk_sta3
pass. No_msgs_in_outbuf_at_sta3
medium = tk...msg...for...sta2

get_tk_sta2
pass. No_msgs_in_outbuf_at_sta2
medium = tk...msg...for...sta1

get_tk_sta1
pass. No_msgs_in_outbuf_at_sta1
medium = tk...msg...for...sta3

get_tk_sta3
pass. No_msgs_in_outbuf_at_sta3
medium = tk...msg...for...sta2

get_tk_sta2
pass. No_msgs_in_outbuf_at_sta2
medium = tk...msg...for...sta1

get_tk_sta1
pass. No_msgs_in_outbuf_at_sta1
medium = tk...msg...for...sta3

get_tk_sta3
pass. No_msgs_in_outbuf_at_sta3
medium = tk...msg...for...sta2

get_tk_sta2
pass. No_msgs_in_outbuf_at_sta2
medium = tk...msg...for...sta1

get_tk_sta1
pass. No_msgs_in_outbuf_at_sta1
medium = tk...msg...for...sta3

get_tk_sta3
pass. No_msgs_in_outbuf_at_sta3
medium = tk...msg...for...sta2

get_tk_sta2
pass. No_msgs_in_outbuf_at_sta2
medium = tk...msg...for...sta1

get_tk_sta1
pass. No_msgs_in_outbuf_at_sta1
medium = tk...msg...for...sta3

get_tk_sta3
pass. No_msgs_in_outbuf_at_sta3
medium = tk...msg...for...sta2

get_tk_sta2
pass. No_msgs_in_outbuf_at_sta2
medium = tk...msg...for...sta1

get_tk_sta1
pass. No_msgs_in_outbuf_at_sta1
medium = tk...msg...for...sta3

get_tk_sta3
pass. No_msgs_in_outbuf_at_sta3
medium = tk...msg...for...sta2

get_tk_sta2
pass. No_msgs_in_outbuf_at_sta2
medium = tk...msg...for...sta1

get_tk_sta1
pass. No_msgs_in_outbuf_at_sta1
medium = tk...msg...for...sta3

get_tk_sta3
pass. No_msgs_in_outbuf_at_sta3
medium = tk...msg...for...sta2

get_tk_sta2
pass. No_msgs_in_outbuf_at_sta2
medium = tk...msg...for...sta1

get_tk_sta1
pass. No_msgs_in_outbuf_at_sta1
medium = tk...msg...for...sta3

get_tk_sta3
pass. No_msgs_in_outbuf_at_sta3
medium = tk...msg...for...sta2

get_tk_sta2
pass. No_msgs_in_outbuf_at_sta2
medium = tk...msg...for...sta1

get_tk_sta1
pass. No_msgs_in_outbuf_at_sta1
medium = tk...msg...for...sta3

get_tk_sta3
pass. No_msgs_in_outbuf_at_sta3
medium = tk...msg...for...sta2

get_tk_sta2
pass. No_msgs_in_outbuf_at_sta2
medium = tk...msg...for...sta1

get_tk_sta1
pass. No_msgs_in_outbuf_at_sta1
medium = tk...msg...for...sta3

get_tk_sta3
pass. No_msgs_in_outbuf_at_sta3
medium = tk...msg...for...sta2

get_tk_sta2
pass. No_msgs_in_outbuf_at_sta2
medium = tk...msg...for...sta1

get_tk_sta1
pass. No_msgs_in_outbuf_at_sta1
medium = tk...msg...for...sta3

get_tk_sta3
pass. No_msgs_in_outbuf_at_sta3
medium = tk...msg...for...sta2

get_tk_sta2
pass. No_msgs_in_outbuf_at_sta2
medium = tk...msg...for...sta1

get_tk_sta1
pass. No_msgs_in_outbuf_at_sta1
medium = tk...msg...for...sta3

get_tk_sta3
pass. No_msgs_in_outbuf_at_sta3
medium = tk...msg...for...sta2

get_tk_sta2
pass. No_msgs_in_outbuf_at_sta2
medium = tk...msg...for...sta1

get_tk_sta1
pass. No_msgs_in_outbuf_at_sta1
medium = tk...msg...for...sta3

get_tk_sta3
pass. No_msgs_in_outbuf_at_sta3
medium = tk...msg...for...sta2

get_tk_sta2
pass. No_msgs_in_outbuf_at_sta2
medium = tk...msg...for...sta1

get_tk_sta1
pass. No_msgs_in_outbuf_at_sta1
medium = tk...msg...for...sta3

get_tk_sta3
pass. No_msgs_in_outbuf_at_sta3
medium = tk...msg...for...sta2

get_tk_sta2
pass. No_msgs_in_outbuf_at_sta2
medium = tk...msg...for...sta1

get_tk_sta1
pass. No_msgs_in_outbuf_at_sta1
medium = tk...msg...for...sta3

get_tk_sta3
pass. No_msgs_in_outbuf_at_sta3
medium = tk...msg...for...sta2

get_tk_sta2
pass. No_msgs_in_outbuf_at_sta2
medium = tk...msg...for...sta1

get_tk_sta1

100 Transitions completed.

YOUR BUS RIDE IS OVER!!!

APPENDIX C

SAMPLE PROGRAM TRACE

```
Script started on Tue May 29 00:25:59 1990
nps-cs [[1]] tkbus


GOING FOR A RIDE WITH THE TOKEN BUS PROTOCOL!


Initial conditions are:
  All stations are in state0
  Token holding time (THT) is set at 4 msgs per station
  medium = tk...msg...for...sta3


get_tk_sta3
xmit_msg_from_sta3
 medium = data...msg...from...sta3...for...sta1
rcv_msg_at_sta1
sta1_ready
 medium = .....clear
more_data_in_outbuf_at_sta3
xmit_msg_from_sta3
 medium = data...msg...from...sta3...for...sta1
rcv_msg_at_sta1
sta1_ready
 medium = .....clear
more_data_in_outbuf_at_sta3
xmit_msg_from_sta3
 medium = data...msg...from...sta3...for...sta1
rcv_msg_at_sta1
sta1_ready
 medium = .....clear
pass_tk_from_sta3
 outbuf...is...empty
 medium = tk...msg...for...sta2
```

get_tk_sta2
xmit_msg_from_sta2
 medium = data...msg...from...sta2...for...sta3
rcv_msg_at_sta3
sta3_ready
 medium = ...clear
pass_tk_from_sta2
 outbuf...is...empty
 medium = tk...msg...for...sta1

get_tk_sta1
xmit_msg_from_sta1
 medium = data...msg...from...sta1...for...sta2
rcv_msg_at_sta2
sta2_ready
 medium = ...clear
more_data_in_outbuf_at_sta1
xmit_msg_from_sta1
 medium = data...msg...from...sta1...for...sta2
rcv_msg_at_sta2
sta2_ready
 medium = ...clear
more_data_in_outbuf_at_sta1
xmit_msg_from_sta1
 medium = data...msg...from...sta1...for...sta2
rcv_msg_at_sta2
sta2_ready
 medium = ...clear
more_data_in_outbuf_at_sta1
xmit_msg_from_sta1
 medium = data...msg...from...sta1...for...sta2
rcv_msg_at_sta2
sta2_ready
 medium = ...clear
pass_tk_from_sta1
 THT...exceeded
 medium = tk...msg...for...sta3

```
get_tk_sta3
xmit_msg_from_sta3
  medium = data...msg...from...sta3...for...sta1
rcv_msg_at_sta1
sta1_ready
  medium = .....clear
more_data_in_outbuf_at_sta3
xmit_msg_from_sta3
  medium = data...msg...from...sta3...for...sta1
rcv_msg_at_sta1
sta1_ready
  medium = .....clear
more_data_in_outbuf_at_sta3
xmit_msg_from_sta3
  medium = data...msg...from...sta3...for...sta1
rcv_msg_at_sta1
sta1_ready
  medium = .....clear
pass_tk_from_sta3
  outbuf...is...empty
  medium = tk...msg...for...sta2


get_tk_sta2
xmit_msg_from_sta2
  medium = data...msg...from...sta2...for...sta3
rcv_msg_at_sta3
sta3_ready
  medium = .....clear
pass_tk_from_sta2
  outbuf...is...empty
  medium = tk...msg...for...sta1


get_tk_sta1
xmit_msg_from_sta1
  medium = data...msg...from...sta1...for...sta2
rcv_msg_at_sta2
sta2_ready
  medium = .....clear
more_data_in_outbuf_at_sta1
xmit_msg_from_sta1
  medium = data...msg...from...sta1...for...sta2
```

```
rcv_msg_at_sta2
sta2_ready
  medium = .....clear
more_data_in_outbuf_at_sta1
xmit_msg_from_sta1
  medium = data...msg...from...sta1...for...sta2
rcv_msg_at_sta2
sta2_ready
  medium = .....clear
more_data_in_outbuf_at_sta1
xmit_msg_from_sta1
  medium = data...msg...from...sta1...for...sta2
rcv_msg_at_sta2
sta2_ready
  medium = .....clear
pass_tk_from_sta1
  THT...exceeded
  medium = tk...msg...for...sta3


get_tk_sta3
xmit_msg_from_sta3
  medium = data...msg...from...sta3...for...sta1
rcv_msg_at_sta1
sta1_ready
  medium = .....clear
more_data_in_outbuf_at_sta3
xmit_msg_from_sta3
  medium = data...msg...from...sta3...for...sta1
rcv_msg_at_sta1
sta1_ready
  medium = .....clear
more_data_in_outbuf_at_sta3
xmit_msg_from_sta3
  medium = data...msg...from...sta3...for...sta1
rcv_msg_at_sta1
sta1_ready
  medium = .....clear
pass_tk_from_sta3
  outbuf...is...empty
  medium = tk...msg...for...sta2
```

get_tk_sta2
xmit_msg_from_sta2
 medium = data...msg...from...sta2...for...sta3
rcv_msg_at_sta3
sta3_ready
 medium = ...clear
pass_tk_from_sta2
 outbuf...is...empty
 medium = tk...msg...for...sta1

get_tk_sta1
xmit_msg_from_sta1
 medium = data...msg...from...sta1...for...sta2
rcv_msg_at_sta2
sta2_ready
 medium = ...clear
more_data_in_outbuf_at_sta1
xmit_msg_from_sta1
 medium = data...msg...from...sta1...for...sta2

rcv_msg_at_sta2
sta2_ready
 medium = ...clear
more_data_in_outbuf_at_sta1
xmit_msg_from_sta1
 medium = data...msg...from...sta1...for...sta2
rcv_msg_at_sta2
sta2_ready
 medium = ...clear
more_data_in_outbuf_at_sta1
xmit_msg_from_sta1
 medium = data...msg...from...sta1...for...sta2
rcv_msg_at_sta2
sta2_ready
 medium = ...clear
pass_tk_from_sta1
 THT...exceeded
 medium = tk...msg...for...sta3

get_tk_sta3
xmit_msg_from_sta3
 medium = data...msg...from...sta3...for...sta1
rcv_msg_at_sta1
sta1_ready
 medium = ...clear
more_data_in_outbuf_at_sta3
xmit_msg_from_sta3
 medium = data...msg...from...sta3...for...sta1
rcv_msg_at_sta1
sta1_ready
 medium = ...clear
more_data_in_outbuf_at_sta3
xmit_msg_from_sta3
 medium = data...msg...from...sta3...for...sta1
rcv_msg_at_sta1
sta1_ready
 medium = ...clear
pass_tk_from_sta3
 outbuf...is...empty
 medium = tk...msg...for...sta2

get_tk_sta2
xmit_msg_from_sta2
 medium = data...msg...from...sta2...for...sta3
rcv_msg_at_sta3
sta3_ready
 medium = ...clear
pass_tk_from_sta2
 outbuf...is...empty
 medium = tk...msg...for...sta1

get_tk_sta1
xmit_msg_from_sta1
 medium = data...msg...from...sta1...for...sta2
rcv_msg_at_sta2
sta2_ready
 medium = ...clear
more_data_in_outbuf_at_sta1
xmit_msg_from_sta1
 medium = data...msg...from...sta1...for...sta2

```
rcv_msg_at_sta2
sta2_ready
  medium = .....clear
more_data_in_outbuf_at_sta1
xmit_msg_from_sta1
  medium = data...msg...from...sta1...for...sta2
rcv_msg_at_sta2

 100 Transitions completed.



 YOUR BUS RIDE IS OVER!!!
```

REFERENCES

[1] Bochmann, Gregor V. and Gecsei, Jan, "A unified Method for the Specification and Verification of Protocols," *Information Processing*, 1977, North-Holland.

[2] Brinksma, Ed, "A Tutorial on LOTOS," *Protocol Specification, Testing and Verification V*, North-Holland, 1985.

[3] Brown, G.M., Gouda, M.G., and Miller, R.M., "Block Acknowledgement: Redesigning the Window Protocol," *Proceedings of the ACM SIGCOMM Symposium*, 1989, Austin TX.

[4] Choi, Tat Y., "Formal Techniques for Specification, Verification, and Construction of Communication Protocols," *IEEE Communications*, 23(10), October 1985.

[5] Gallatin, T. and Khatib, H. A., "Token Bus vs. CSMA/CD for Broadband Backbone LANs," *Proceedings of the IEEE Conference on Systems Design and Networks*, Santa Clara, CA, April 1989.

[6] Gburzynski, P. and Rudnicki, P. "A Better-than-Token Protocol with Bounded Delay Time for Ethernet type LANs," *Symposium on the Simulation of Computer Networks*, Dept of Computer Science, University of Alberta, Edmonton, Alberta, Canada , August 1987.

[7] Institute of Electrical and Electronics Engineers, Inc., *IEEE Standard 802.3, Carrier Sense Multiple Access with Collision Detection Access Method and Physical Layer Specifications*, 1985.

[8] Institute of Electrical and Electronics Engineers, Inc., *IEEE Standard 802.4, Token Bus Access Method and Physical Layer Specifications*, 1985.

[9] Institute of Electrical and Electronics Engineers, Inc., *IEEE Standard 802.5, Token Ring Access Method and Physical Layer Specifications*, 1985.

[10] Johnson, M. J., "Proof that Timing Requirements of the FDDI Token Ring Protocol are Satisfied," *IEEE Transactions on Communications Vol Com-35, No. 6*, June 1987.

[11] Linn, R. J., "The Features and Facilities of Estelle: a formal description technique based upon an extended finite state machine model," *Protocol Specification, Testing and Verification V*, North-Holland, 1985.

[12] Lundy, G. M., *Systems of Communicating Machines: A Model for Communication Protocols*, Ph.D. Thesis, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1988.

[13] Lundy, G. M., "Specification and Analysis of the Token Bus Protocol Using Systems of Communicating Machines," *Proceedings of the IEEE Conference on Systems Design and Networks*, Santa Clara, CA, May 8-10, 1990.

[14] Lundy, G. M. and Akyildiz, Ian F., "A Formal Model of the FDDI Network Protocol," Department of Computer Science, Naval Postgraduate School, Monterey, Ca, June 1990.

[15] Lundy, G. M. and Miller, Raymond E., *Specification and Analysis of a CSMA/CD Protocol Using Systems of Communicating Machines*, submitted to the 15th Annual Conference on Local Computer Networks, IEEE Computer Society, Minneapolis, MN, 1990.

[16] Lundy, G. M., and Miller, Raymond E., *A Variable Window Protocol Specification and Analysis*, Eighth International Symposium on Protocol Specification, Testing and Verification, Atlantic City, NJ, June 7-10, 1988.

[17] Lundy, G. M. and Luqi, "Specification of a Token Ring Protocol Using Systems of Communicating Machines," *Proceedings of the IEEE Conference on Systems Design and Networks*, Santa Clara, CA, April 1989.

[18] Raiche, Carl, *A Specification and Analysis of the IEEE Token Ring Protocol*, M.S. Thesis, Department of Computer Science, Naval Postgraduate School, Monterey, CA, 1989.

[19] Ross, F. E., "An Overview of FDDI: The Fiber Distributed Data Interface," *IEEE Journal on Selected Areas in Communications*, September 1989

[20] Rudin, Harry, "An Informal Overview of Formal Protocol Specification," *IEEE Communications*, 23(3), March 1985.

[21] Shoch, J., Dalal, Y., Redell, D., Crane, R., "The Ethernet," *Lecture Notes in Computer Science, (184), Local Area Networks: An Advanced Course*, Glasgow, July 1983.

[22] Tanenbaum, A., *Computer Networks, Second Edition*, Engelwood Cliffs, N.J., Prentice Hall, Inc., 1988.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center — 2
   Cameron Station
   Alexandria, Virginia 22304-6145

2. Library, Code 0142 — 2
   Naval Postgraduate School
   Monterey, California 93943-5002

3. Department Chairman, Code 52 — 2
   Department of Computer Science
   Naval Postgraduate School
   Monterey, California 93943-5000

4. Curricular Officer, Code 37 — 2
   Computer Technology
   Naval Postgraduate School
   Monterey, California 93943-5000

5. Superintendent — 2
   Computer Science Department, Chauvenet Hall
   United States Naval Academy
   Annapolis, Maryland 21402

6. Professor G. M. Lundy — 3
   Computer Science Department, Code 52Ln
   Naval Postgraduate School
   Monterey, California 93943-5000

7. Professor Man-Tak Shing, — 2
   Computer Science Department, Code 52Sh
   Naval Postgraduate School
   Monterey, California 93943-5000

8. Professor L. Williamson — 2
   Computer Science Department, Code 52Wi
   Naval Postgraduate School
   Monterey, California 93943-5000